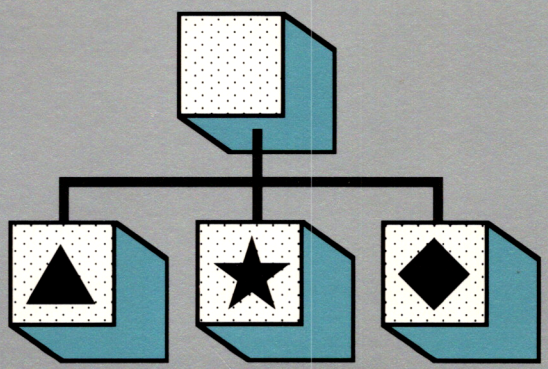
TURBO PASCAL

Volume 3

TSRs and More

Object Professional™

A library of over 100 powerful object types containing over 1000 methods. Window object types let you use multiple overlapping resizeable windows providing: ■ Menus ■ Pick lists ■ Data entry ■ Text editing and much more. Data object types and system-oriented routines are included too!



TURBO Power

Object Professional

User's Manual

Volume 3
TSRs and More

Copyright (c) 1989-1991, TurboPower Software
All Rights Reserved

Second Edition December 1991

P.O. Box 49009
Colorado Springs, CO 80949-9009

Technical Support: 719-260-6641
Fax: 719-260-7151
Sales: 800-333-4160
9 a.m. to 5 p.m. Mountain Time, Monday-Friday

CompuServe Mail: 76004,2611
CompuServe Support: PCVENB Section 6

Trademarks Mentioned

TurboPower Software and the distinctive TurboPower logo are trademarks of TurboPower Software.

Object Professional is a trademark of TurboPower Software.

Turbo Professional is a registered trademark of Sunny Hill Software, used under license to TurboPower Software.

Turbo Pascal, Turbo Assembler, Turbo Debugger, Turbo Database Toolbox, Turbo Editor Toolbox, SideKick, SideKickPlus, and SuperKey are trademarks or registered trademarks of Borland International.

DesqView is a trademark of Quarterdeck Office Systems.

Double DOS is a trademark of SoftLogic Solutions.

OmniView is a trademark of Sunny Hill Software.

Periscope is a trademark of The Periscope Company.

Hercules is a registered trademark of Hercules Computer Technology, Inc.

dBase is a trademark of Ashton-Tate.

Lotus is a trademark of Lotus Development Corporation.

WordStar is a registered trademark of MicroPro International Corporation.

MS-DOS, OS/2, Windows, CodeView, and Macro Assembler are trademarks of Microsoft Corporation.

IBM PC, XT, AT, PS/2, and TopView are registered trademarks of International Business Machines Corporation.

Table of Contents

Volume 1: Screen Management

1. Introduction 1-1
   - A. System Requirements 1-4
   - B. Files Supplied 1-5
   - C. Installation 1-7
   - D. Organization of the Manual 1-14
   - E. Getting Help with POPHELP 1-17
   - F. DESKPOP: Desktop Manager 1-24
   - G. Purchase Agreement 1-45

2. Principles of Organization 2-1

3. Screen and Keyboard 3-1
   - A. Overview 3-1
   - B. OPCRT: Enhanced CRT Unit 3-2
   - C. OPCOLOR : Color Names 3-49
   - D. OPMOUSE: Mouse Routines 3-52
   - E. OPCMD: Command Processing 3-82

4. Window Objects 4-1
   - A. Overview 4-1
   - B. OPFRAME: Frame Objects 4-8
   - C. OPWINDOW: Window Objects 4-55
   - D. OPPICK: Pick Lists 4-186
   - E. OPDIR: Directory and Path Lists 4-241
   - F. OPMENU: Menu Systems 4-297

Index IN-1

Volume 2: Data Input/Output

4. Window Objects (continued) 4-361
   - G. OPHELP: Help Systems 4-361
   - H. Utility and Sample Programs 4-420

5. Text Editors and File Browsers 5-1
   - A. Overview 5-1
   - B. OPMEMO: Basic Text Editor 5-2
   - C. OPEDITOR: Advanced Text Editor 5-30
   - D. OPBROWSE: File Browser 5-56
   - C. Sample Programs 5-79

6. Data Entry 6-1
A. Overview 6-1
B. OPABSFLD: Abstract Field Definitions 6-4
C. OPFIELD: Data Entry Fields 6-15
D. OPFEDIT: Field Editors 6-52
E. OPSELECT: Field Selection 6-55
F. OPENTRY: Data Entry Screens 6-110
G. OPEDIT: Advanced Line Editor 6-176
H. OPSEDIT: Simple Line Editor 6-197
I. OPDATE: Date and Time Variables 6-206
J. Utility and Sample Programs 6-228

7. Printing 7-1
A. Overview 7-1
B. OPPRNLOW: Physical Level Printer Support 7-3
C. OPPRINT: Logical Level Printer Support 7-27
D. OPDEVICE: Hardware Specific Printer Support 7-57
E. OPFORM: Printed Forms 7-66
F. Sample Programs 7-113

Index IN-1

Volume 3: TSRs and More

8. Data Objects 8-1
A. Overview 8-1
B. OPROOT: Object-Oriented Data Structures 8-2
C. OPLARRAY: Object-Oriented Large Arrays 8-154

9. String Handling 9-1
A. Overview 9-1
B. OPSTRING: String Manipulation 9-2
C. OPASCIIZ: Long ASCIIZ Strings 9-35
D. OPSTRDEV: Formatted Strings 9-49

10. DOS/BIOS Calls 10-1
A. Overview 10-1
B. OPDOS: Assorted DOS/BIOS Calls 10-3
C. OPEXEC: Swapping Exec Manager 10-32
D. OPENHKBD: Enhanced Keyboard Interface 10-42

11. Interrupts and TSRs 11-1
A. Overview 11-1
B. OPINT: ISR Management 11-4
C. OPTSR: TSR Management 11-13
D. Overview of Swappable TSRs 11-34
E. OPSWAP1: Swappable TSRs, Part 1 11-47
F. OPSWAP: Swappable TSRs, Part 2 11-56
G. OP8087: 8087 Support 11-58
H. Sample Programs 11-64

12. Memory Allocation 12-1
A. Overview 12-1
B. OPEMS: EMS Memory Management 12-3
C. OPEXTMEM: Extended Memory Management 12-12

13. Macros 13-1
A. Overview 13-1
B. OPMACRO: Keyboard Macros 13-3
C. OPMACED: Macro Editor 13-15
D. OPREPLAY: Macro Playback 13-18
E. Sample Programs 13-23

14. Miscellaneous 14-1
A. Overview 14-1
B. OPBCD: High-precision Reals 14-2
C. OPCLONE: Program Customization 14-16
D. OPERRHAN: Runtime Error Recovery 14-25
E. OPINLINE: Assorted Inline Macros 14-28
F. OPSORT: In-memory Sort Routine 14-45

15. Appendices 15-1
A. What's New in Object Professional 1.0 15-1
B. Unit Dependencies 15-4
C. Object Hierarchy 15-7
D. Error Codes 15-10
E. Using Overlays with Object Professional 15-14
F. Converting Turbo Professional Applications 15-16
G. Compiling Large Programs 15-24
H. For Further Reading 15-26

Procedure and Function Reference PR-1

Index IN-1

Object Quick Reference

Volume 1: Screen Management

3. Screen and Keyboard

ColorSet 3-48
CommandProcessor 3-95
CommandPacker 3-115

4. Window Objects

ScreenRect 4-10
AbstractFrame 4-31
Frame 4-42
RawWindow 4-62
AbstractWindow 4-69
StackWindow 4-118
CommandWindow 4-143
WindowStack 4-132
LoadableColorSet 4-162
PackedWindow 4-166
VirtScreen 4-175
PickList 4-208
DirList 4-251
PathList 4-285
MenuItemNode 4-314
Menu 4-323

Volume 2: Data Input/Output

4. Window Objects (continued)

AbstractHelpWindow 4-384
ScrollingHelpWindow 4-415

5. Text Editors and File Browsers

Memo 5-10
MemoFile 5-22
TextEditor 5-41
Browser 5-64

6. Data Entry
SelectField 6-18
EntryField 6-64
AbstractSelector 6-64
Selector 6-90
ScrollingSelector 6-103
EntryScreen 6-144
ScrollingEntryScreen 6-169
LineEditor 6-188
SimpleLineEditor 6-201

7. Printing
BasePrinter 7-10
DosPrinter 7-19
BiosPrinter 7-20
Printer 7-38
PrintField 7-88
PrintedForm 7-92
Report 7-109

Volume 3: TSRs and More

8. Data Objects
PointerStack 8-9
StaticQueue 8-21
SingleListNode 8-33
SingleList 8-34
DoubleListNode 8-43
DoubleList 8-44
CircularList 8-49
BitSet 8-52
LargeBitSet 8-64
StringDict 8-71
StringSet 8-80
StringArray 8-82
Root 8-99
IdStream 8-100
DosIdStream 8-120
BufIdStream 8-123
MemIdStream 8-125
DirEntry 8-132
Library 8-137
MemLibrary 8-148
AbstractArray 8-161

14. Miscellaneous
Cloner 14-19

8. Data Objects

A. Overview

Although the units to be described in this chapter are object-oriented, they represent a different category of objects than the other ones previously described. The data objects aren't associated specifically with writing user interfaces, for one thing; they provide basic building blocks that you can use directly in an application or from which you can construct other objects of your own.

The bulk of this chapter is devoted to describing the OPROOT unit. OPROOT contains a dozen or so general purpose data structures--stacks, lists, queues, etc. It also implements the **Object Professional** stream manager, which provides object-oriented file I/O capabilities for all of the other objects. And, because OPROOT is used by so many other **Object Professional** units, it is a convenient repository for a variety of constants--error codes, unit codes, error message strings, object type codes, and object version codes.

Also described here is the OPLARRAY unit, which offers several ways to create one- or two-dimensional arrays of arbitrary data type. Because of the object-oriented design of this unit, you can access array data exactly the same way whether it is stored in normal memory, expanded memory, or on disk.

B. OPROOT: Object-Oriented Data Structures

OPROOT provides a collection of low-level objects for managing many classic data structures. These include stacks, lists, queues, bit sets, and others. By virtue of their object-oriented design, these data structures are *extensible*. You can extend them to manage linked lists of your own data type, for example. We use these objects ourselves throughout **Object Professional**: the WindowStack is derived from the general purpose PointerStack; the list of headers for a window is a special case of a DoubleList; the set of selected items in a multiple choice PickList is an instance of a BitSet; and so on. Counting on these general-purpose data structures in your own programs will save you time, promote re-use of object code, and probably give you capabilities that you wouldn't think of writing for special purpose data structures.

The following hierarchy shows the data-management objects provided by OPROOT.

```
[Root]
  PointerStack
  StaticQueue
  SingleListNode
  SingleList
  DoubleListNode
  DoubleList
    CircularList
  BitSet
    LargeBitSet
  StringDict
    StringSet
  StringArray
```

A brief overview of each object here will allow you to choose the appropriate one for your needs. The following sections provide detailed descriptions of each one.

Root is the abstract parent of all **Object Professional** objects that support streams. We'll return to it in the section entitled "Streams" below.

As the name implies, a PointerStack stores pointers in a stack. It provides methods to push, pop, and peek into the stack. Typically the pointers will point to instances of an object derived from the Root type, although this is not required.

A StaticQueue implements one form of the queue data structure, which provides methods to add to end, take from front, and so on. The StaticQueue stores a specified number of bytes for each element in the queue. It is designed using a circular buffer of a specified capacity.

A SingleListNode is an abstract type that forms the basis of a singly-linked list. Its only data field is a Next pointer. Actual list nodes are derived from this type.

A SingleList provides all the capabilities you would expect from a singly-linked list. A given list stores nodes of one or more types derived from SingleListNode.

DoubleListNode and DoubleList implement doubly-linked lists. Each list node contains four bytes more data than SingleListNode to store the backward link. DoubleLists are preferred over SingleLists when the list must be traversed in either direction.

A CircularList is a special case of a DoubleList. A request for the next element beyond the end of a CircularList returns the first element, in contrast to the DoubleList, which returns nil.

A BitSet is a data structure that efficiently stores arrays of boolean values by packing 8 booleans into each byte of memory. It offers methods to set and clear bits, check the value of a given bit, scan from bit to bit, and many more.

While the standard BitSet is limited to storing 65534 values, the LargeBitSet can store 8 million or as many as will fit into memory.

The StringDict implements a hash table data structure. Each element in a StringDict stores a string (the dictionary entry name) and a four-byte value associated with the string. The object offers methods to add, test for membership, and remove elements from the dictionary. The four-byte data field holds an application-specific value, perhaps a token number associated with a parser or even a pointer to a larger record describing the dictionary element.

A StringSet offers a limited subset of the functionality of the StringDict. The StringSet omits the four-byte data value. Hence, it provides operations analogous to normal Pascal sets but with the set elements formed by strings.

The StringArray provides an efficient data structure for storing and accessing large arrays of strings. Instead of allocating the size of the longest string for each string, it packs all of the strings end to end, and manages an index to find each string quickly.

Streams

OPROOT includes another category of objects which is the basis for a very powerful and general capability of **Object Professional**: Streams. Streams are introduced at length in section 6 of this chapter. Briefly put, however, a stream is a sequence of bytes storing almost any kind of data, much like an untyped file. In **Object Professional** streams are used primarily to store the data from object instances. A stream can be general-purpose–storing objects of arbitrary type–because it depends on the objects themselves to perform two important tasks: writing and reading themselves to and from the stream. Each object that provides such capabilities has methods called Store and Load. OPROOT's stream manager counts on these methods when performing all object-oriented input and output.

The complexity of the stream objects in OPROOT is increased substantially in order to deal with an important practical issue. If OPROOT referred directly to the Load and Store methods of each object type that it could read and write, Turbo Pascal's smart linker would pull in the object code for every one of those methods. That's obviously undesirable, especially if a given stream will contain only one type of object. To avoid this problem, OPROOT provides a number of methods for "registering" object types with a stream. In this way, the stream manager becomes aware of the types it can read and write, and avoids linking in excessive code. It also means that the stream manager can read and write objects that weren't even in existence when it was first developed; that's important so you can use it with your own derived types.

So far, we've described streams only as a sequence of bytes. Another important difference between streams and untyped files is that, at the most abstract level, streams don't assume any connection with a DOS disk file. By keeping the objects abstract as long as possible, OPROOT is able to generate a number of useful variations on the general concept. Using OPROOT streams, objects can be stored sequentially in a DOS disk file, they can be linked directly in a program's EXE file, or they can be managed via random-access libraries on disk or linked into the code. Because **Object Professional** provides stream support for all its important objects, this means that you can link window images into your program, manage libraries of data entry screens, or use a collection of prewritten printer drivers, among many other possibilities, all via the OPROOT stream objects.

You may be aware that Borland provides a stream capability as an example unit with Turbo Pascal 5.5. The OPROOT streams, while similar in concept and implementation to Borland's, go much further toward making this capability useful in production software. Specifically, OPROOT streams offer the following advantages:

- dynamic registration of user types
- object version control
- stream support for pointer types
- loading of static instances as well as dynamic
- in-memory as well as disk file streams
- random-access stream libraries

The meaning of these terms will become clearer as you read about the stream objects in this chapter.

The following hierarchy shows the stream management objects provided by OPROOT.

```
Root
  DirEntry
IdStream
  DosIdStream
    BufIdStream
      Library
  MemIdStream
    MemLibrary
```

Root is an abstract parent for almost all **Object Professional** object types. Its main purpose is to assure that the virtual method table (VMT) link resides in the first two bytes of each object instance. This is essential for proper functioning of the stream manager. Deriving all objects from a common ancestor provides other advantages (discussed in the Root object's section).

DirEntry is a low-level object used for managing entries in a stream library.

IdStream is an abstract type that serves to organize all of the other stream objects. It provides methods for low-level read and write and for registering data types with the stream manager. The "Id" part of the name serves to distinguish these streams from Borland's implementation and to indicate that each object in the stream identifies itself by an "Identification" code.

The DosIdStream associates an abstract stream with a DOS file. This object provides methods for opening, creating, and closing stream files. And of course it inherits all of the read, write, and registration methods from the IdStream.

A BufIdStream adds buffering to the DosIdStream. A memory buffer is allocated to store stream information temporarily on its way to or from disk. This improves stream performance, especially when the stream is composed of many small data blocks.

A Library manages a collection of individual object streams in a single DOS file. Each object in a library is identified by name; the Library manager can seek directly to a given object and read or write it. The Library provides methods to add, delete, update, rename, and pack entries.

MemIdStream and MemLibrary are just like DosIdStream and Library, with one important difference: the streams are stored in memory instead of within a disk file. Programs typically will take advantage of this feature by using Borland's BINOBJ utility to create an OBJ file from a stream file and then linking the OBJ file directly into the EXE file. In this way, an application can avoid the need for a separate data file.

General Purpose Constants, Types, and Variables

OPROOT is a convenient place to store several categories of declarations that are referred to by many other **Object Professional** Units. All of the object-oriented units already USE OPROOT in order to access the Root and IdStream types, so no additional dependencies are generated. The general purpose declarations are grouped into the following categories:

- object type codes
- object version codes
- pointer codes
- unit codes
- error codes
- error message strings
- error procedure type
- constructor status variable

The first three categories are closely associated with streams; these constants are described in section 6 of this chapter. The others are described in section 8, Error Handling.

1. Stacks: The PointerStack

Object Professional's primary implementation of a stack is the PointerStack object. As you might expect, a PointerStack manages a stack of pointers. However, the data types managed by the PointerStack can be either more general or more specific than the name implies. More generally, a PointerStack (with appropriate use of typecasting) can store any data type with size ranging from one to four bytes. Even so, the PointerStack is used to best advantage when it stores pointers to objects: specifically, those objects derived from the **Object Professional** Root type. For example, the WindowStack (in the OPWINDOW unit) is derived from a PointerStack, and it stores pointers to the various window object types derived from Root.

A stack, of course, is a data type that supports at least the operations of pushing and popping. (It is a LIFO--last in, first out--data structure.) A stack can be implemented in at least two different ways: as a simple array, or as a special form of linked list. The array approach is advantageous because indexing into the stack is more efficient and memory usage and fragmentation are less. The linked list approach is nice because the stack can grow indefinitely, within available memory.

The PointerStack is implemented as an array of pointers, but with a twist: it knows how to dynamically resize itself, thus providing the positive characteristics of both alternatives. Within reasonable limits, you don't need to know in advance how many elements will be stored on a PointerStack. Even so, to avoid heap fragmentation it's a good idea to initialize the stack for the actual maximum capacity expected.

The PointerStack provides all the methods you would expect from a stack, including Push, Pop, and Clear. It also lets you Peek at items hidden in the middle of the stack.

The following example demonstrates the major features of the PointerStack object.

```
program PointerStackExample;  {EXPSTACK.PAS}
uses
  OpRoot;
type
  {An object for manipulating Name and Age}
  NameDataPtr = ^NameData;
  NameData =
    object(Root)
      Name : String[20];
      Age : Byte;
      constructor Init;
      procedure Display;
    end;
var
  NamePtr : NameDataPtr;
  NamePtrStack : PointerStack;
  I : Integer;
  Status : Word;

  {----- NameData Methods ------------------}
  constructor NameData.Init;
  begin
    WriteLn;
    Write('Name:  ');
    ReadLn(Name);
    Write('Age:   ');
    ReadLn(Age);
  end;
```

```
  procedure NameData.Display;
  begin
    WriteLn;
    Writeln(Name,' ',Age);
  end;

begin
  {Create a stack for storing up to 5 pointers}
  if not NamePtrStack.Init(5) then begin
    WriteLn('Failed to Init PointerStack,  Status = ', InitStatus);
    Halt;
  end;

  {Create and Push three NameData objects}
  for I := 1 to 3 do begin
    New(NamePtr, Init);
    NamePtrStack.Push(NamePtr);
    Status := NamePtrStack.GetStatus;
    if Status <> 0 then begin
      WriteLn('Failed to push NameData, Status = ', Status);
      Halt;
    end;
  end;

  {Peek at the very first object pushed}
  NamePtr := NameDataPtr(NamePtrStack.Peek(1));
  NamePtr^.Display;

  {Peek at the top item, without actually popping it}
  NamePtr := NameDataptr(NamePtrStack.PeekTop);
  NamePtr^.Display;

  {Now pop each NameDataPtr off and display it, the 4th call will fail}
  for I := 1 to 4 do begin
    NamePtr := NameDataPtr(NamePtrStack.Pop);
    if NamePtr = nil then begin
      WriteLn('Stack is empty');
      Halt;
    end;
    NamePtr^.Display;
  end;

  NamePtrStack.Done;
end.
```

The example initializes a PointerStack capable of storing 5 pointers. (To do so, it allocates heap space. If the stack were to grow beyond 5 the PointerStack would deallocate the initial buffer and allocate a new one twice the initial size.) The Init call will fail if there is not enough heap space.

Three instances of a simple object type, NameData, derived from Root, are allocated and pushed onto the stack. We check the status of each Push by calling the GetStatus method, which returns zero when all is well. The Push calls won't fail unless the stack size is forced to expand and there is not enough heap space.

Next, we peek at the first element pushed onto the stack. The first element always has an index of 1. If there were no elements on the stack, Peek would return nil. Note that a typecast is required to assign the result of Peek to the NamePtr variable. This is inherent to object-oriented programming.

We also peek at the top element on the stack with a call to PeekTop.

Next, we pop the elements from the stack. In this example we pop one more element than was pushed, and compare the returned pointer to nil to detect when the stack is empty.

Finally, we call the PointerStack's Done destructor to deallocate the heap space used by the PointerStack. This final step is optional; it's important only if free heap space is needed and the program will continue after the need for the stack is over.

Constants

```
MaxStackSize = 65520 div SizeOf(RootPtr);
```

The maximum number of pointers that can ever be stored in a PointerStack. This works out to 16380 pointers.

```
InitStackSize = 8;
```

The size of the stack created if a call to Push is made and no stack has previously been allocated.

Types

```
PointerStackPtr = ^PointerStack;
PointerStack =
  object(Root)
    ...
  end.
```

Data types defining a PointerStack object and a pointer to a PointerStack.

Declaration
```
procedure Clear;
```

Purpose
Remove all elements from the stack.

Description
This routine simply sets the internal stack pointer to zero. None of the stack elements are disposed, nor is the stack itself deallocated.

Example
```
NamePtrStack.Clear;
```
Removes all pointers from NamePtrStack.

Declaration
```
destructor Done; virtual;
```

Purpose
Deallocate the stack.

Description
Call this routine when the stack is no longer needed. It deallocates the heap space used to manage the stack. None of the stack elements are disposed.

Examples
```
NamePtrStack.Done;
```
Deallocates heap space used by NamePtrStack.
```
Dispose(StackP, Done);
```
Disposes of a dynamic stack object and deallocates the heap space used by the stack.

See Also
Init

Declaration
```
procedure Error(Code : Word); virtual;
```

Purpose
Report an error.

Description
Applications do not call this routine directly. When a PointerStack method detects an error, it calls the Error method to report it. The default implementation of this method simply assigns the error code to an internal status variable. An object derived from PointerStack may override the Error method to provide different error handling behavior.

See Also
GetStatus

GetStatus / Init

Declaration

```
function GetStatus : Word;
```

Purpose

Return latest status code and reset internal status to zero.

Description

Each PointerStack maintains an internal variable that stores the current status of the stack. GetStatus provides access to this variable. Like IoResult, GetStatus clears the internal status variable after it returns the current value.

When no error has occurred, GetStatus returns 0. Otherwise it returns an error code as specified in Appendix D. The only expected error is epFatal+ecOutOfMemory.

Example

```
New(NamePtr, Init);
NamePtrStack.Push(NamePtr);
Status := NamePtrStack.GetStatus;
if Status <> 0 then begin
  WriteLn('Failed to push NameData. Status = ', Status); Halt;
end;
```

After pushing a new pointer onto the stack, we call GetStatus to check for errors. Since calling the GetStatus function resets the error value, we first move the value into a Status variable (of type Word).

If we know we will never push more pointers than we allocated space for (with Init), then we don't really need to check for errors here. However, if we exceed the requested number of pointers, Push will try to allocate additional heap space, which introduces the possibility of an "out of memory" error. Only Init, Push, and Resize can fail with this kind of error.

Declaration

```
constructor Init(Max : Word);
```

Purpose

Allocate a stack of pointers that can manage up to Max items.

Description

This constructor allocates 4*Max bytes of heap space in addition to that required when the PointerStack is a dynamic object itself. The stack pointer is initialized to 0 (no elements are on the stack).

If the value of Max is zero, Init initializes its internal data values (all to zero), but allocates no heap space. This is sometimes useful when constructing objects that depend on the PointerStack. It is safe to call PointerStack.Done after this form of call to Init.

PointerStacks typically are used to store pointers to objects derived from Root. However, with appropriate typecasting they can store any data type whose size is four bytes or less.

If the constructor fails, it returns a boolean result of False or a pointer result of nil, depending on whether the PointerStack is defined as a static or dynamic instance. Error status is returned in the OPROOT global variable InitStatus.

Example

```
if not NamePtrStack.Init(5) then begin
  WriteLn('Failed to init PointerStack. Status = ', InitStatus); Halt;
end;
```

Initializes a pointer stack that holds up to five pointers without reallocation.

See Also

Done Resize

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a PointerStack from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to PointerStack.Store. Load allocates heap space for the stack, and pushes any items onto the stack that were active at the time the PointerStack was stored.

Load uses the polymorphic IdStream.GetPtr call to read these pointers. GetPtr automatically calls the Load constructor of the object the pointer points to, but only if the object was previously registered with the stream. Hence, Load can be used only for PointerStacks that hold pointers to objects derived from Root, or that were empty at the time the stack was stored.

If Load fails, the constructor will return False and an error code will be stored in the global variable InitStatus.

The stream registration procedure for a PointerStack is PointerStackStream.

Example

```
uses
  OpRoot;
const
  otNameData = 1000;          {Name object type}
  veNameData = 0;             {Name object version}
type
  NameDataPtr = ^NameData;
  NameData =
    object(Root)
      Name : String[20];
      Age : Byte;
      procedure Display;
      procedure Store(var S : IdStream);
      constructor Load(var S : IdStream);
    end;
var
  NamePtr : NameDataPtr;
  NamePtrStack : PointerStack;
  NPS : BufIdStream;

  procedure NameData.Display;
  begin
    Writeln(Name,' ',Age);
  end;

  procedure NameData.Store(var S : IdStream);
  begin
    S.WriteString(Name);
    S.Write(Age, SizeOf(Age));
  end;

  constructor NameData.Load(var S : IdStream);
  begin
    Root.Init;
    Name := S.ReadString;
```

```
    S.Read(Age, SizeOf(Age));
    if S.PeekStatus <> 0 then
      Fail;
  end;

begin
  {Open the buffered stream file}
  NPS.Init('PTRSTACK.STM', SOpen, 1024);

  {Register the object types contained in the stack}
  NPS.RegisterType(otNameData, veNameData, TypeOf(NameData),
                   @NameData.Store, @NameData.Load);

  {Load the PointerStack}
  if not NamePtrStack.Load(NPS) then begin
    WriteLn('Load error ', InitStatus);
    Halt;
  end;

  {Close the stream}
  NPS.Done;

  {Now pop each NameDataPtr off and display it}
  repeat
    NamePtr := NameDataPtr(NamePtrStack.Pop);
    if NamePtr <> nil then
      NamePtr^.Display;
  until NamePtr = nil;
end.
```

This example instantiates a pointer stack from the stream created in the Store example. Not only does the example allocate the stack, it also reads and allocates all of the objects stored in that stream (in this example, three instances of NameData).

An important part of opening a stream is registering the data types it contains. See section 6 for more information.

An effect similar to calling Load may be achieved with the IdStream.Get method. IdStream.Get is often desirable because it is polymorphic; that is, it automatically determines the appropriate Load constructor to call based on the stream's data. To use Get, the stream must first be created using IdStream.Put. Then the Load call in the example above may be replaced with

```
NPS.RegisterHier(PointerStackStream);
NPS.Get(NamePtrStack);
Status := NPS.GetStatus;
if Status <> 0 then begin
  WriteLn('Load error, ', Status);
  Halt;
end;
```

See Also

Store
IdStream.GetPtr
IdStream.RegisterType

Declaration
```
function Peek(Index : Word) : RootPtr;
```

Purpose
Return stack element at position Index.

Description
The first element pushed onto a PointerStack gets an index of 1. Additional elements get successively higher index numbers. Peek provides access to elements anywhere on a stack by referencing their index numbers. Peek doesn't modify the stack.

If the Index passed to Peek is zero or a number higher than the topmost element on the stack, Peek returns nil.

Example
```
NamePtr := NameDataPtr(NamePtrStack.Peek(1));
NamePtr^.Display;
```

Gets the first pointer pushed onto the stack. Peek returns a pointer of type RootPtr, which must be typecast to NameDataPtr.

See Also
Pop

Declaration
```
function PeekTop : RootPtr;
```

Purpose
Return pointer currently at the top of stack.

Description
This call is equivalent to Peek(SP). It returns the element currently at the top of the stack. If there are no elements currently on the stack, PeekTop returns nil. PeekTop doesn't modify the stack.

Example
```
NamePtr := NameDataPtr(NamePtrStack.PeekTop);
NamePtr^.Display;
```

Gets the top pointer (the last pointer that was pushed) without modifying the stack. We typecast the returned pointer to NameDataPtr.

See Also
Pop

Poke / Pop

Declaration

```
procedure Poke(P : RootPtr; Index : Word);
```

Purpose

Directly set a stack element.

Description

This method allows replacing any specified element on the stack with a new value. Calling Poke doesn't change the stack pointer. If the Index value is zero or larger than the current stack capacity, Poke does nothing. Poke does allow changing the stack at values above the current stack pointer.

Example

```
New(NamePtr, Init);
NamePtrStack.Poke(NamePtr, 1);
```

Instantiate a new Name object, then poke a pointer to that object into the very bottom of the stack (as though it were the first pointer we pushed).

See Also

Push

Declaration

```
function Pop : RootPtr;
```

Purpose

Pop the topmost pointer off the stack.

Description

Returns the pointer at the top of the stack and decrements the stack pointer by 1. If there is nothing on the stack, Pop returns nil.

Example

```
for I := 1 to 3 do begin
  NamePtr := NameDataPtr(NamePtrStack.Pop);
  if NamePtr = nil then begin
    WriteLn('Stack is empty');
    Halt;
  end;
  NamePtr^.Display;
end;
```

Retrieves three pointers from the stack in last-in-first-out (LIFO) fashion. Because Pop returns a pointer of type RootPtr, we must typecast it to a NameDataPtr when assigning its result to another variable.

See Also

PeekTop

Declaration

```
procedure Push(P : RootPtr);
```

Purpose

Push P onto the stack.

Description

Increments the stack pointer by 1 and stores the pointer P into that stack location.

If the current capacity of the stack is exceeded, Push attempts to double the stack's size by allocating a new heap block, copying the existing stack contents to the new block, then deallocating the old block. If insufficient heap space is available in this case, Push does nothing. This kind of error can be detected by calling GetStatus after the call to Push.

Examples

```
for I := 1 to 3 do begin
  New(NamePtr, Init);
  NamePtrStack.Push(NamePtr);
  Status := NamePtrStack.GetStatus;
  if Status <> 0 then begin
    WriteLn('Failed to push NameData, Status = ', Status);
    Halt;
  end;
end;
```

Creates and pushes three instances of NameData on the NamePtrStack.

```
Name.Init;
NamePtrStack.Push(Name);
```

Incorrect! Push expects a pointer to a Root object, not the object itself. For a static instance, the appropriate statement would be

```
NamePtrStack.Push(@Name);
```

See Also

Pop
Poke
Resize
Size

Resize / Size / SP

Declaration

```
function Resize(Max : Word) : Boolean;
```

Purpose

Expand or shrink stack to hold Max pointers.

Description

This routine first allocates heap space for Max pointers (4*Max bytes). It then copies any existing stack elements, up to the capacity of the new stack, onto the new stack. If the new stack is smaller than the old and the old stack had more pointers than will now fit, the top of the stack is truncated. Resize can be used to make the stack larger or smaller than the current size.

Resize returns True if it succeeded. It will return False only if there is insufficient heap space for the new stack.

Example

```
if not NamePtrStack.Init(5) then begin
  WriteLn('Failed to init PointerStack. Status = ', InitStatus); Halt;
end;
{...push pointers...}
if not NamePtrStack.Resize(50) then begin
  WriteLn('Failed to resize PointerStack'); Halt;
end;
```

The NamePtrStack is initialized with only five slots. Later in the program we resize it to handle 50 pointers.

Declaration

```
function Size : Word;
```

Purpose

Return the size of the stack.

Description

This function returns the current capacity of the stack, not the number of elements currently pushed onto it. The size is set when the stack is initialized, when Resize is called, or when the stack automatically expands itself as the result of a Push statement.

The size returned is the number of pointer-sized entries, not the number of bytes.

Example

```
WriteLn('NamePtrStack can hold ', NamePtrStack.Size, ' entries.');
```

Reports the maximum number of entries that NamePtrStack can hold.

Declaration

```
function SP : Word;
```

Purpose

Return the current top-of-stack index.

Description

If nothing is on the stack, SP returns 0. Otherwise it returns the index of the element last pushed onto the stack.

Example

```
WriteLn('The current top of the stack is ', NamePtrStack.SP);
```

Reports the current top of the stack, which is also the number of entries that are currently stacked.

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a stack in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the PointerStack to the stream S. The image includes the capacity of the stack, the current number of elements pushed onto it, and the images associated with each pushed pointer.

Store uses the polymorphic IdStream.PutPtr call to write these pointers. PutPtr automatically calls the Store method of the object the pointer points to, but only if the object was previously registered with the stream. Hence, Store can be used only for PointerStacks that hold pointers to objects derived from Root, or that are empty at the time the stack is stored.

To check for errors that may have occurred during the Store, call the stream's GetStatus function as shown in the example below.

The stream registration procedure for a PointerStack is PointerStackStream.

Example

```
uses
  OpRoot;
const
  otNameData = 1000;          {Name object type code}
  veNameData = 0;             {Name object version}
type
  NameDataPtr = ^NameData;
  NameData =
    object(Root)
      Name : String[20];
      Age : Byte;
      constructor Init;
      procedure Store(var S : IdStream);
      constructor Load(var S : IdStream);
    end;
var
  NamePtr : NameDataPtr;
  NamePtrStack : PointerStack;
  NPS : BufIdStream;
  I : Integer;
  Status : Word;

  constructor NameData.Init;
  begin
    WriteLn;
    Write('Name:  ');
    ReadLn(Name);
    Write('Age:   ');
    ReadLn(Age);
  end;

  procedure NameData.Store(var S : IdStream);
  begin
    S.WriteString(Name);
    S.Write(Age, SizeOf(Age));
  end;
```

```
  constructor NameData.Load(var S : IdStream);
  begin
    Root.Init;
    Name := S.ReadString;
    S.Read(Age, SizeOf(Age));
    if S.PeekStatus <> 0 then
      Fail;
  end;

begin
  {Create a stack for storing up to 5 pointers}
  if not NamePtrStack.Init(5) then begin
    WriteLn('Failed to Init PointerStack,  Status = ', InitStatus);
    Halt;
  end;

  {Create and Push three NameData objects}
  for I := 1 to 3 do begin
    New(NamePtr, Init);
    NamePtrStack.Push(NamePtr);
  end;

  {Write the pointer stack to a stream}
  NPS.Init('PTRSTACK.STM', SCreate, 1024);
  NPS.RegisterType(otNameData, veNameData, TypeOf(NameData),
                   @NameData.Store, @NameData.Load);
  NamePtrStack.Store(NPS);
  Status := NPS.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error ', Status);
    Halt;
  end;

  {Close the stream}
  NPS.Done;
end.
```

This example creates a pointer stack of three NameData objects and writes that stack to a stream. Writing the stack to a stream entails writing the size and current top-of-stack values, then indirectly calling the Store procedure to write out the objects being pointed to. That's why we register the NameData object.

An effect similar to calling Store may be achieved with the IdStream.Put method. IdStream.Put is often desirable because it is polymorphic; that is, it automatically determines the appropriate Store method to call based on the stream's data. When a stream is written with Put, the same stream must be read with Get. Get can then determine automatically which Load constructor to call. To use Put, the Store call in the example above may be replaced with

```
NPS.RegisterHier(PointerStackStream);
NPS.Put(NamePtrStack);
```

See Also

Load IdStream.PutPtr

2. Queues: The StaticQueue

The StaticQueue object implements another classic data structure, the Queue. A queue is a structure that provides operations for adding a new item to the end of the queue and taking an item from the front of the queue, just like the line at a movie ticket window. Another term for this data structure is a FIFO, for first-in, first-out.

StaticQueue is a specific implementation of this general idea. It manages a circular queue of a predetermined size. If all slots in the queue are filled, StaticQueue will either reject a new entrant or push the member at the other end out of the queue, depending on an option setting.

Unlike the PointerStack, the StaticQueue isn't limited to storing pointers. It stores data items of any size, up to a total queue size of 64K bytes. The primary disadvantage of this approach is that when StaticQueues are read from streams, the stream manager cannot use a polymorphic Load to read each queue element. Instead it simply reads each queue element as a byte image of specified size.

The StaticQueue provides a versatile set of methods for managing this data structure. It can add or remove elements from the tail, read or peek at elements at the front, clear the queue, and store and load queues from streams.

The StaticQueue was first written to support the "topics previously shown" stack in the AbstractHelpWindow object. In that case, it was used more like a stack than a queue; this serves to indicate the flexibility of the StaticQueue.

The following example program demonstrates the main features of StaticQueue.

```
program StaticQueueExample;  {EXSQUEUE.PAS}
uses
  OpRoot;
var
  Name : String;
  NameQueue : StaticQueue;
  I : Integer;
begin
  {Create a StaticQueue capable of holding 5 strings, each up to 255 chars}
  if not NameQueue.Init(5, SizeOf(Name), True) then begin
    WriteLn('Failed to create StaticQueue,  Status = ', InitStatus);
    Halt;
  end;

  {Create three Names, add to tail of the queue}
  for I := 1 to 3 do begin
    Write('Enter name: ');
    ReadLn(Name);
    NameQueue.PushTail(Name);
    if NameQueue.GetStatus <> 0 then begin
      WriteLn('Failed to Push Name');
      Halt;
    end;
  end;
```

```
  {Retrieve elements from the head of the queue}
  for I := 1 to 3 do begin
    NameQueue.PopHead(Name);
    if NameQueue.GetStatus <> 0 then
      WriteLn ('Failed to Pop Name')
    else
      WriteLn(Name);
  end;

  {Dispose of the queue}
  NameQueue.Done;
end.
```

The example creates a queue capable of holding 5 elements, each one of which can contain a full 255 character string. Heap space totaling 1280 bytes (=5*256) is allocated by the Init constructor for the queue's data storage. The third parameter to Init is a Boolean called RejectIfFull. Because it's set to True here, the queue will reject additional elements when it is already full and an attempt is made to add another.

Three elements are added to the tail of the queue with PushTail and then the elements are retrieved from the head with PopHead. Calls to GetStatus check for errors.

Finally, the heap space used by the queue is deallocated with the call to Done.

Types

```
StaticQueuePtr = ^StaticQueue;
StaticQueue =
  object(Root)
    ...
  end;
```

The data types defining a StaticQueue object and a pointer to a StaticQueue.

Declaration
```
procedure Clear;
```

Purpose
Remove all elements from the queue.

Description
The queue's head and tail pointers are set equal, a condition that indicates the queue is empty. Clear also resets the internal queue status variable to 0.

Example
```
NameQueue.Clear;
```
Removes all elements from the queue.

Declaration
```
destructor Done; virtual;
```

Purpose
Deallocate space for the queue.

Description
Call this routine when the queue is no longer needed. It deallocates the heap space used to manage the queue.

Example
```
NameQueue.Done;
```
Deallocates the heap space used by NameQueue.

Declaration
```
function Elements : Word;
```

Purpose
Return the number of elements currently in the queue.

Description
If the queue has just been created or cleared, this returns zero. The largest number ever returned will be one less than the total number of elements allocated by the queue's constructor call. (One element position must be used to indicate that the queue is empty or full.)

Example
```
WriteLn('NameQueue current has ', NameQueue.Elements, ' elements');
```
Reports the number of elements currently stored in NameQueue.

See Also
Init

Error / GetStatus

Declaration

```
procedure Error(Code : Word); virtual;
```

Purpose

Assign error code.

Description

Applications do not call this routine directly. When a StaticQueue method detects an error, it calls the Error method to report it. The default implementation of this method simply assigns the error code to an internal status variable. An object derived from StaticQueue may override the Error method to provide different error handling behavior.

Declaration

```
function GetStatus : Word;
```

Purpose

Return latest status code and reset internal status to zero.

Description

Each StaticQueue maintains an internal variable that stores the current status of the queue. GetStatus provides access to this variable. Like IoResult, GetStatus clears the internal status variable after it returns the current value.

When no error has occurred, GetStatus returns 0. Otherwise it returns an error code as specified in Appendix D. Possible error codes are

```
epNonFatal+ecQueueFull     attempt to add element when queue is full
epNonFatal+ecQueueEmpty    attempt to read element when queue is empty
```

Example

```
NameQueue.PopHead(Name);
Status := NameQueue.GetStatus;
if Status <> 0 then
  WriteLn ('Failed to Pop Name, Status = ', Status)
else
  WriteLn(Name);
```

After popping the head element, we call GetStatus to check for errors. Since referencing GetStatus resets it, we have to move the status value to one of our variables if we wish to report it.

Declaration

```
constructor Init(NumElements, ElementSize : Word; RejectIfFull : Boolean);
```

Purpose

Allocate space for static queue.

Description

This constructor allocates NumElements*ElementSize bytes of heap space in addition to that required if the StaticQueue is a dynamic object itself. The product of the first two parameters must be no larger than 65521 bytes or the constructor will fail. The maximum capacity of the queue is actually one less than NumElements because one spot must be reserved to indicate that the circular queue is empty or full.

If the value of NumElements is zero, Init initializes its internal data values (all to zero), but allocates no heap space. This is sometimes useful when constructing objects that depend on a StaticQueue. It is safe to call StaticQueue.Done after this form of call to Init.

The RejectIfFull parameter determines the queue's behavior when the queue is full and a request is made to add an element. If RejectIfFull is True, the queue will do nothing but return an error (epNonFatal+ecQueueFull). If RejectIfFull is False, the element will be added but the element at the front of the queue will be "pushed out of line" and lost.

If the constructor fails, it returns a boolean result of False or a pointer result of nil, depending on whether the StaticQueue is defined as a static or dynamic instance. Error status is returned in the OPROOT global variable InitStatus.

Example

```
if not NameQueue.Init(5, SizeOf(Name), True) then begin
  WriteLn('Failed to create StaticQueue,  Status = ', InitStatus);
  Halt;
end;
```

Initializes a static queue to hold up to five Names. In this example, we want requests to PushTail to be rejected if the queue is full.

A static queue can store virtually any kind of data; we are using it to store strings.

See Also

Done

Load

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a static queue from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to StaticQueue.Store. Load allocates heap space for the queue, and adds any items to the queue that were active at the time the StaticQueue was stored.

Load uses the equivalent of a BlockRead call to read each element. Unlike the PointerStack, it does not make any form of polymorphic call to read the elements.

If Load fails, the constructor will return False and an error code will be stored in the global variable InitStatus.

The stream registration procedure for a StaticQueue is StaticQueueStream.

Example

```
uses
  OpRoot;
var
  Name : String;
  NameQueue : StaticQueue;
  NQS : BufIdStream;
  I : Integer;
begin
  {Initialize NameQueue from a stream}
  NQS.Init('NQUEUE.STM', SOpen, 1024);
  if not NameQueue.Load(NQS) then begin
    WriteLn('Load error, ', InitStatus);
    Halt;
  end;
  NQS.Done;

  {Retrieve elements from the queue}
  for I := 1 to 3 do begin
    NameQueue.PopHead(Name);
    if NameQueue.GetStatus <> 0 then
      WriteLn('Failed to Pop Name')
    else
      WriteLn(Name);
  end;
end.
```

Loads NameQueue from the stream created in the Store example. This builds the queue and loads all the existing elements into the new queue.

An effect similar to calling Load may be achieved with the IdStream.Get method. See PointerStack.Load for further information.

See Also

Store IdStream.Get

Declaration

```
procedure PeekHead(var Element);
```

Purpose

Look at the element at the front of the queue.

Description

Reads the element at the front of the queue into the Element parameter, but does not remove it from the queue. If the queue is empty, Element is filled with zeros and an error code (epNonFatal+ecQueueEmpty) is reported. The status can be checked by calling GetStatus.

Example

```
NameQueue.PeekHead(Name);
if NameQueue.GetStatus <> 0 then begin
  WriteLn ('Queue is empty')
else
  WriteLn(Name);
```

Returns the oldest Name on the queue without changing the queue. Since this operation could return an "empty queue" error, we need to check for that condition.

See Also

GetStatus PopHead

Declaration

```
procedure PeekTail(var Element);
```

Purpose

Look at the element at the tail of the queue.

Description

Reads the element at the tail of the queue into the Element parameter, but does not remove it from the queue. If the queue is empty, Element is filled with zeros and an error code (epNonFatal+ecQueueEmpty) is reported. The status can be checked by calling GetStatus.

Example

```
NameQueue.PeekTail(Name);
if NameQueue.GetStatus <> 0 then begin
  WriteLn ('Queue is empty')
else
  WriteLn(Name);
```

Returns the last Name pushed onto the queue without changing the queue. Since this operation could return an "empty queue" error, we need to check for that condition.

See Also

GetStatus PopHead

PopHead / PopTail

Declaration

```
procedure PopHead(var Element);
```

Purpose

Remove an element from the front of the queue.

Description

Reads the element at the front of the queue into the Element parameter, and removes it from the queue. If the queue is empty, Element is filled with zeros and an error code (epNonFatal+ecQueueEmpty) is reported. The status can be checked by calling GetStatus.

Classic queue behavior is obtained by adding elements to the tail with PushTail and removing them from the front with PopHead.

Be sure to pass a variable of the appropriate type to PopHead. If the variable is smaller than the ElementSize used to initialize the queue, PopHead will wipe out memory. It has no way of knowing whether the variable is large enough.

Example

```
for I := 1 to 3 do begin
  NameQueue.PopHead(Name);
  if NameQueue.GetStatus <> 0 then WriteLn('Queue is empty')
  else WriteLn(Name);
end;
```

Retrieves three Names from the queue (starting at the head) in the same order in which they were pushed. Since this operation could return an "empty queue" error, we need to check for that condition.

See Also

GetStatus PeekHead

Declaration

```
procedure PopTail(var Element);
```

Purpose

Remove an element from the tail of the queue.

Description

Reads the element at the tail of the queue into the Element parameter, and removes it from the queue. If the queue is empty, Element is filled with zeros and an error code (epNonFatal+ecQueueEmpty) is reported. The status can be checked by calling GetStatus.

The StaticQueue may be used as a static stack by adding items with PushTail and removing them with PopTail.

Example

```
for I := 1 to 3 do begin
  NameQueue.PopTail(Name);
  if NameQueue.GetStatus <> 0 then WriteLn ('Queue is empty')
  else WriteLn(Name);
end;
```

Retrieves the last three Names pushed onto the queue. The last Name pushed is the first one retrieved.

See Also

GetStatus PushTail
PeekTail

Declaration

```
procedure PushTail(var Element);
```

Purpose

Add an element to the tail of the queue.

Description

PushTail provides the only method for adding an element to the queue. If the queue is already full, the element will be added only if the RejectIfFull parameter passed to Init was False. In that case, the item at the head of the queue will be "pushed out of line" and lost. Otherwise, an error (epNonFatal+ecQueueFull) will be reported.

PushTail reads ElementSize bytes from the untyped parameter Element.

Example

```
NameQueue.Init(5, SizeOf(Name), True);
for I := 1 to 3 do begin
  Write('Enter name: ');
  ReadLn(Name);
  NameQueue.PushTail(Name);
  if NameQueue.GetStatus <> 0 then begin
    WriteLn('Queue is full');
    Halt;
  end;
end;
```

Pushes three new Names onto the tail of NameQueue. How the queue was initialized is important here. NameQueue was initialized to reject attempts to add items to a full queue. Therefore, after each attempt to add an item, we may need to check for a queue-full failure.

If the queue were initialized with RejectIfFull = False, the queue would always make room for new items by throwing away the item at the head of the queue when required. In that case, it would be unnecessary to call GetStatus after each call to PushTail.

See Also

Init

Store

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a static queue to a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the StaticQueue to the stream S. The image includes the capacity of the queue, the current number of elements in it, and the byte image associated with each element.

To check for errors that may have occurred during the Store, call the stream's GetStatus function as shown in the example below.

The stream registration procedure for a StaticQueue is StaticQueueStream.

Example

```
uses OpRoot;
var  Name : String;
     NameQueue : StaticQueue;
     NQS : BufIdStream;
     I : Integer;
     Status : Word;
begin
  {Create a StaticQueue}
  if not NameQueue.Init(5, SizeOf(Name), True) then begin
    WriteLn('Failed to make StaticQueue. Status = ', InitStatus); Halt;
  end;
  {Create three Names, push onto the StaticQueue}
  for I := 1 to 3 do begin
    Write('Enter name: ');
    ReadLn(Name);
    NameQueue.PushTail(Name);
    if NameQueue.GetStatus <> 0 then begin
      WriteLn('Failed to Push Name'); Halt;
    end;
  end;
  {Store the elements to a stream}
  NQS.Init('NQUEUE.STM', SCreate, 1024);
  NameQueue.Store(NQS);
  Status := NQS.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error, ', Status); Halt;
  end;
  NQS.Done;
  NameQueue.Done;
end.
```

Creates three Names and adds them to a static queue. The entire queue is then written to a stream. See Load for an example of how to restore this data.

An effect similar to calling Store may be achieved with the IdStream.Put method. See PointerStack.Store for further information.

See Also

IdStream.Put Load

3. Linked Lists

Linked lists are perhaps the most ubiquitous data structure in all of structured programming. We use them heavily in **Object Professional**: to manage window headers in OPWINDOW, entry fields in OPENTRY, print blocks in OPFORM, and menu items in OPMENU, among others. To assure that an appropriate list type is available for all needs, OPROOT supplies three different list implementations: singly linked, doubly linked, and circular. This section describes the associated objects.

A linked list manages a sequence of data elements. Fundamentally, it offers methods to add or delete elements from the sequence and to step from one element to the next. Because the size of a linked list isn't defined in advance, it is ideal for problems where the number of data elements isn't known at the outset.

The singly linked list offers one-way transportation: it is efficient at moving from one element to the next element, but not to the previous. The doubly linked list uses one more pointer of data storage to make traversing the list equally efficient in both directions.

The circular list is a special form of doubly linked list. When the end of a doubly linked list is reached, one can go no further. With a circular list, the end joins seamlessly with the beginning.

Linked lists fit particularly well with object oriented programming's concepts of inheritance and polymorphism. The linked list nodes provided here don't really contain any data. The data they do include is the bare minimum to allow management of the list. The payoff comes when you derive your own objects from the provided linked list node types. Your nodes inherit the data links from their ancestor and, more importantly, they inherit all the list management methods from OPROOT's linked list objects. If you've ever written an application that depended heavily on lists and found yourself writing the same list management code over and over, you'll certainly appreciate what OOP is doing for you in this situation.

OOP-style linked lists even have a capability that you probably haven't used before: One list can contain nodes of differing types. OPENTRY and OPFORM take full advantage of this concept when storing data fields of many types.

The following program example demonstrates the major features of a singly linked list.

```
program SingleListExample;  {EXSLIST.PAS}
uses
  OpRoot;
type
  {An object for manipulating Name and Age, derived from SingleListNode}
  NameDataPtr = ^NameData;
  NameData =
    object(SingleListNode)
      Name : String[20];
      Age  : Byte;
      constructor Init;
      procedure Display;
    end;

var
  NamePtr : NameDataPtr;
  NameList : SingleList;
  I : Integer;
```

```
  {----- NameData Methods ------------------}
  constructor NameData.Init;
  begin
    if not SingleListNode.Init then
      Fail;
    WriteLn;
    Write('Name:  ');
    ReadLn(Name);
    Write('Age:   ');
    ReadLn(Age);
  end;

  procedure NameData.Display;
  begin
    WriteLn;
    Writeln(Name, ' ', Age);
  end;

begin
  {Create a SingleList}
  NameList.Init;

  {Create three NameData objects, append to the List}
  for I := 1 to 3 do begin
    New(NamePtr, Init);
    NameList.Append(NamePtr);
  end;

  {Insert a NameData object at the front of the List}
  New(NamePtr, Init);
  NameList.Insert(NamePtr);

  {Get the Head, then traverse the entire List}
  NamePtr := NameDataPtr(NameList.Head);
  while NamePtr <> nil do begin
    NamePtr^.Display;
    NamePtr := NameDataPtr(NameList.Next(NamePtr));
  end;

  {Traverse the List again, this time removing AND disposing each object}
  NamePtr := NameDataPtr(NameList.Head);
  while NamePtr <> nil do begin
    NameList.Delete(NamePtr);
    NamePtr := NameDataPtr(NameList.Head);
  end;

  {Done with the list in full}
  NameList.Done;
end.
```

Because the NameData object is derived from a SingleListNode, it inherits the next-field link which is essential to the list manager's operation. The NameList in the example is just an instance of a SingleList. The list manager doesn't care that it is managing nodes consisting of NameData objects rather than some other type. In other cases, it may be appropriate to derive a new type from SingleList. This is typically done only when special methods for accessing the list are required.

Note that NameData.Init starts out by calling SingleListNode.Init. Doing so initializes the next-field link of the node to nil and also calls the constructor of the Root object from which SingleListNode is derived. After that, NameData.Init initializes its own data fields.

The main program starts by initializing the linked list instance NameList with a call to the Init constructor of SingleList. This call marks the list as completely empty. Next, three instances of the NameData type are allocated on the heap and initialized. These three instances are appended to the end of the linked list. A fourth instance is initialized and inserted at the front of the linked list.

The example gets the head of the list and traverses from node to node displaying each item until it reaches the end, when the Next method returns nil.

You'll notice that a typecast is required to convert the results of Head and Next to the NameDataPtr type. This typecasting takes some getting used to. The OOPDEMOS.DOC file that comes with Turbo Pascal 5.5 discusses the issue briefly, and and we'll put the rules into our own words here.

When you're making an assignment from an instance of one type to an instance of a derived type, a typecast is required. In this example, we're assigning an instance of SingleNodePtr to one of NameDataPtr. The typecast tells the compiler, in effect, that even though a SingleNode variable doesn't contain all the information that a NameData node does, you're taking responsibility for the assignment. You *know* that the list contains NameData nodes, so the assignment is safe.

An assignment in the reverse direction--from a NameDataPtr to a SingleNodePtr in the example--doesn't require a typecast. That's because the NameData variable is guaranteed to have all the information that a SingleNode does, because it's derived from it. We're effectively making such an assignment when we pass the NamePtr variable as a parameter to Next.

The final loop in the example shows how to dispose of items in a linked list. The Delete method unlinks the specified item from the list, makes a polymorphic call to the Done destructor of the item itself, and then deallocates the heap space of the item node. The polymorphic call allows the item to clean up any of its own business. The NameData object doesn't need to do anything special, so it doesn't implement its own Done method.

Another method, Remove, can be used when you want to unlink an item from a list but not dispose of it.

The call to NameList.Done is redundant in this example, but it is shown for completeness. The Done method does the same thing as the preceding loop--it unlinks and disposes of all the items in the list.

An example program for a doubly linked list would be almost identical to the one just described. SingleListNode and SingleList would be replaced by DoubleListNode and DoubleList, but that's the only required change.

The following sequence might be added to demonstrate traversing the list in the reverse direction:

```
{Get the Tail, then traverse then entire List backward}
NamePtr := NameDataPtr(NameList.Tail);
while NamePtr <> nil do begin
  NamePtr^.Display;
  NamePtr := NameDataPtr(NameList.Prev(NamePtr));
end;
```

This calls the Tail and Prev methods, but is otherwise identical to the single list example.

A CircularList example would also be quite similar. Since the CircularList is derived from the DoubleList, the NameData node type in the example would still be derived from the DoubleListNode. (There is no such thing as a CircularListNode.) The main difference for a circular list is how one traverses the entire list. A loop like this is required:

```
var
  SavePtr : NameDataPtr;

SavePtr := NamePtr;
repeat
  NamePtr^.Display;
  NamePtr := NameDataPtr(NameList.Next(NamePtr));
until NamePtr = SavePtr;
```

The traversal can start at any point in the list; it will always make a full loop. Here it is starting at the item NamePtr which was previously initialized when a new item was added.

The objects for linked lists are described hereafter in the following order:

```
SingleListNode
SingleList
DoubleListNode
DoubleList
CircularList
```

Types

```
SingleNodePtr = ^SingleListNode;
SingleListNode =
  object(Root)
    ...
  end;
SingleListPtr = ^SingleList;
SingleList =
  object(Root)
    ...
  end;
```

Data types for managing singly linked lists.

```
DoubleNodePtr = ^DoubleListNode;
DoubleListNode =
  object(Root)
    ...
  end;
DoubleListPtr = ^DoubleList;
DoubleList =
  object(Root)
    ...
  end;
```

Data types for managing doubly linked lists.

```
CircularListPtr = ^CircularList;
CircularList =
  object(DoubleList)
    ...
  end;
```

Data types for managing circular lists.

Note: The CircularList inherits most of its methods from DoubleList. Only those methods that it overrides are re-documented. See the entries for DoubleList for the remaining methods.

Declaration

```
constructor Init;
```

Purpose

Initialize a singly linked list node.

Description

Calls the Root object's Init constructor and then sets the next-field link of the instance to nil. Objects derived from SingleListNode should call SingleListNode.Init before proceeding with their own initialization.

Example

```
constructor NameData.Init;
begin
  if not SingleListNode.Init then
    Fail;
  WriteLn;
  Write('Name:  ');
  ReadLn(Name);
  Write('Age:   ');
  ReadLn(Age);
end;
```

This object, derived from SingleListNode, calls its parent's Init constructor first. As currently implemented, SingleListNode.Init can never fail. Nevertheless, it's a good idea for NameData to check the result and fail its own constructor in case of future enhancements.

See Also

Root.Init

Append / Delete / Done

Declaration
```
procedure Append(P : SingleNodePtr);
```

Purpose
Add element to end of list.

Description
The element pointed to by P is linked to the end of list. If the list is empty, P becomes the first element of the list. If P is nil, Append does nothing.

Example
```
for I := 1 to 3 do begin
  New(NamePtr, Init);
  NameList.Append(NamePtr);
end;
```
Instantiates three new NamePtr objects and appends each new pointer to the NameList linked list.

See Also
Insert Place

Declaration
```
procedure Delete(P : SingleNodePtr);
```

Purpose
Delete existing element in list and dispose of its contents.

Description
The list is first searched to find the element P. If it is not found, Delete does nothing. Otherwise, it unlinks the element from the list, relinks the elements surrounding it, and then calls
```
Dispose(P, Done);
```
which calls P's own destructor and then deallocates the space for the node itself.

Examples
```
NameList.Delete(NameList.Head);
```
Disconnects the current head from the linked list and disposes of it.
```
NameList.Delete(NamePtr);
NamePtr^.Display;
```
Incorrect! The object pointed to by NamePtr has been disposed; it no longer exists.

See Also
Remove

Declaration
```
destructor Done; virtual;
```

Purpose
Destroy list.

Description
Disposes of all items on the list and then destroys the list itself.

Example
```
NameList.Done;
```
Done "walks" down the list, calling the destructor (Done) for each pointer in the list.

Declaration
```
function Head : SingleNodePtr;
```

Purpose
Return pointer to head of list.

Description
If the list is empty, returns the nil pointer. Otherwise, returns a pointer to the first item in the list. Note that the function result must usually be cast to the type of the derived list node.

Example
```
NamePtr := NameDataPtr(NameList.Head);
if NamePtr <> nil then NamePtr^.Display;
```
Retrieves the pointer at the head of the NameList linked list and calls that object's Display method. Since Head returns a SingleNodePtr, we typecast this to a type of NameDataPtr.

Declaration
```
constructor Init;
```

Purpose
Initialize an empty list of items.

Description
Sets the head pointer of the list to nil and the item count to zero.

Examples
```
NameList.Init;
```
Initializes a singly-linked list named NameList. It's not necessary to check the constructor's function result since the constructor just initializes some data fields.
```
if not NameList.Init then {InitStatus has an error code};
```
It's probably a better idea to get in the habit of making constructor calls this way.
```
New(NameListP, Init);
if NameListP = nil then {Error, check InitStatus};
```
Initializes a pointer to a linked list. If NameListP is nil and InitStatus is 0, then there was insufficient heap space to allocate the SingleList itself. If InitStatus is non-zero, then the SingleList constructor itself detected an error.

Declaration
```
procedure Insert(P : SingleNodePtr);
```

Purpose
Insert element at start of list.

Description
The element pointed to by P is inserted at the start of the list. If the list is empty, P becomes the first element of the list. If P is nil, Insert does nothing.

Example
```
New(NamePtr, Init);
NameList.Insert(NamePtr);
```
Instantiates a NamePtr and inserts the pointer at the head of the NameList linked list.

See Also
Append PlaceBefore

Load

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a list from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to SingleList.Store. Load initializes the list and appends any items to it that were active at the time the SingleList was stored.

Load uses the polymorphic IdStream.GetPtr call to read the pointers to each list node. GetPtr automatically calls the Load constructor of the object the pointer points to, but can do so only if the object was previously registered with the stream.

If Load fails, the constructor will return False and an error code will be stored in the global variable InitStatus.

The stream registration procedure for a SingleList is SingleListStream.

Example

```
uses
  OpRoot;
const
  otNameData = 1000;          {Name object type}
  veNameData = 0;             {Name object version}
type
  {An object for manipulating Name and Age, derived from SingleListNode}
  NameDataPtr = ^NameData;
  NameData =
    object(SingleListNode)
      Name : String[20];
      Age  : Byte;
      procedure Display;
      constructor Load(var S : IdStream);
      procedure Store(var S : IdStream);
    end;

var
  NamePtr : NameDataPtr;
  NameList : SingleList;
  NLS : BufIdStream;

  procedure NameData.Display;
  begin
    WriteLn;
    Writeln(Name, ' ', Age);
  end;

  constructor NameData.Load(var S : IdStream);
  begin
    SingleListNode.Init;
    Name := S.ReadString;
    S.Read(Age, SizeOf(Age));
    if S.PeekStatus <> 0 then
      Fail;
  end;
```

```
  procedure NameData.Store(var S : IdStream);
  begin
    S.WriteString(Name);
    S.Write(Age, SizeOf(Age));
  end;

begin
  {Read a linked list in from a stream}
  NLS.Init('SGLLIST.STM', SOpen, 1024);
  NLS.RegisterType(otNameData, veNameData, TypeOf(NameData),
                   @NameData.Store, @NameData.Load);
  if not NameList.Load(NLS) then begin
    WriteLn('Load error, ', InitStatus);
    Halt;
  end;
  NLS.Done;

  {Get the Head, then traverse the entire List, displaying each item}
  NamePtr := NameDataPtr(NameList.Head);
  while NamePtr <> nil do begin
    NamePtr^.Display;
    NamePtr := NameDataPtr(NameList.Next(NamePtr));
  end;
end.
```

Reads a singly-linked list from a stream. The NameList itself is initialized and the Load constructor for each node in the list is called to instantiate those objects. The result in this example is a linked list with three nodes, each pointing to a NamePtr object on the heap.

An effect similar to calling Load may be achieved with the IdStream.Get method. IdStream.Get is often desirable because it is polymorphic; that is, it automatically determines the appropriate Load constructor to call based on the stream's data. To use Get, the stream must first be created using IdStream.Put. Then the Load call in the example above may be replaced with

```
NLS.RegisterHier(SingleListStream);
NLS.Get(NameList);
Status := NLS.GetStatus;
if Status <> 0 then begin
  WriteLn('Load error, ', Status);
  Halt;
end;
```

See Also

IdStream.Get
IdStream.RegisterType
Store

Next / Place

Declaration

```
function Next(P : SingleNodePtr) : SingleNodePtr;
```

Purpose

Returns a pointer to the next node.

Description

Assumes that P is a valid node already in the list. If not, Next will return garbage.

Note that the function result must usually be cast to the type of the derived list node.

Example

```
NamePtr := NameDataPtr(NameList.Head);
while NamePtr <> nil do begin
  NamePtr^.Display;
  NamePtr := NameDataPtr(NameList.Next(NamePtr));
end;
```

First retrieve the pointer at the head of the list. Then, using the Next method, "walk" the linked list, calling each object's Display method.

See Also

DoubleList.Prev

Declaration

```
procedure Place(P, L : SingleNodePtr);
```

Purpose

Place element P into the list after existing element L.

Description

L must be a valid node already on the list. To improve performance, Place does not confirm that this requirement is met.

As a special case, if L is nil, Place inserts the new node P at the start of the list.

If the existing element L is the tail of the list, it's more efficient to call Append.

Examples

```
New(NamePtr, Init);
NameList.Place(NamePtr, NameList.Head);
```

Instantiates a new object and places the object after the current head of NameList. The new pointer is now the second node in the linked list.

```
New(NamePtr, Init);
NameList.Place(NamePtr, OldNamePtr);
```

Incorrect! If OldNamePtr is uninitialized, Place will corrupt memory in some random location.

See Also

PlaceBefore

Declaration
```
procedure PlaceBefore(P, L : SingleNodePtr);
```

Purpose
Place element P into list before existing element L.

Description
L must be a valid node already on the list. PlaceBefore must find the existing element L in order to insert the new element; if L is not found, PlaceBefore does nothing.

As a special case, if L is nil, PlaceBefore inserts the new node P at the start of the list.

There is no way to add an element to the end of the list using PlaceBefore. If this is desired, call Append instead.

Example
```
New(NamePtr, Init);
NameList.PlaceBefore(NamePtr, NameList.Tail);
```

Instantiates a new object and inserts the object before the last node (Tail) of NameList. The new pointer is now the second to last node in the linked list.

See Also
Place

Declaration
```
procedure Remove(P : SingleNodePtr);
```

Purpose
Remove existing element from list without disposing of it.

Description
The node P is unlinked from the list and the elements surrounding it are relinked. The object that P points to remains instantiated, perhaps so that it can be linked into another list or used temporarily before disposal.

If P is not currently a member of the list, Remove does nothing.

Example
```
NameList.Remove(NameList.Head);
```

Disconnects the current head from the linked list but does not dispose of it.

See Also
Delete

Declaration
```
function Size : Word;
```

Purpose
Return the size of the current list.

Description
The size is the number of elements in the list. SingleList updates a counter each time a node is added or removed, and the Size function just returns the value of that counter.

If somehow the list has more than 64K elements, Size will return an invalid result. The list manager will otherwise work fine, however.

Example
```
WriteLn('NameList has ', NameList.Size, ' elements');
```

Displays the current number of elements in the linked list.

Store

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a list in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the SingleList to the stream S. Store uses the polymorphic IdStream.PutPtr call to write an image of each object currently on the list. PutPtr automatically calls the Store method of the object each node pointer points to, but only if the object type was previously registered with the stream.

To check for errors that may have occurred during the Store, call the stream's GetStatus function as shown in the example below.

The stream registration procedure for a SingleList is SingleListStream.

Example

```
uses OpRoot;
const
  otNameData = 1000;          {Name object type}
  veNameData = 0;             {Name object version}
type
  {An object for manipulating Name and Age, descendant from SingleListNode}
  NameDataPtr = ^NameData;
  NameData =
    object(SingleListNode)
      Name : String[20];
      Age  : Byte;
      constructor Init;
      constructor Load(var S : IdStream);
      procedure Store(var S : IdStream);
    end;

var
  NamePtr : NameDataPtr;
  NameList : SingleList;
  NLS : BufIdStream;
  I : Integer;
  Status : Word;

  constructor NameData.Init;
  begin
    WriteLn;
    Write('Name:  ');
    ReadLn(Name);
    Write('Age:   ');
    ReadLn(Age);
  end;

  constructor NameData.Load(var S : IdStream);
  begin
    SingleListNode.Init;
    Name := S.ReadString;
    S.Read(Age, SizeOf(Age));
    if S.PeekStatus <> 0 then Fail;
  end;
```

```
  procedure NameData.Store(var S : IdStream);
  begin
    S.WriteString(Name);
    S.Write(Age, SizeOf(Age));
  end;

begin
  {Create an example list}
  NameList.Init;
  for I := 1 to 3 do begin
    New(NamePtr, Init);
    NameList.Append(NamePtr);
  end;

  {Write the current linked list to a stream}
  NLS.Init('SGLLIST.STM', SCreate, 1024);
  NLS.RegisterType(otNameData, veNameData, TypeOf(NameData),
                   @NameData.Store, @NameData.Load);
  NameList.Store(NLS);
  Status := NLS.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error, ', Status);
    Halt;
  end;
  NLS.Done;
  NameList.Done;
end.
```

This example instantiates a singly-linked list (NameList), adds three elements to it, and writes the list to a stream. The resulting stream contains, in linked list order, every object pointed to by the linked list.

An effect similar to calling Store may be achieved with the IdStream.Put method. IdStream.Put is often desirable because it is polymorphic; that is, it automatically determines the appropriate Store method to call based on the stream's data. When a stream is written with Put, the same stream must be read with Get. Get can then determine automatically which Load constructor to call. To use Put, the Store call in the example above may be replaced with

```
NLS.RegisterHier(SingleListStream);
NLS.Put(NameList);
```

See Also

IdStream.PutPtr
IdStream.RegisterType
Load

Tail

Declaration

```
function Tail : SingleNodePtr;
```

Purpose

Return pointer to tail of list.

Description

If the list is empty, Tail returns nil. Otherwise, it scans from the head of the list to find the last element and returns a pointer to it.

Example

```
NamePtr := NameDataPtr(NameList.Tail);
NamePtr^.Display;
```

Retrieves the pointer at the tail of the NameList linked list (this would be the last pointer Appended) and calls that object's Display method.

See Also

DoubleList.Tail Head

Declaration

```
constructor Init;
```

Purpose

Initialize a doubly linked list node.

Description

Calls the Root object's Init constructor and then sets the next and previous field links of the instance to nil. Objects derived from DoubleListNode should call DoubleListNode.Init before proceeding with their own initialization.

See Also

SingleListNode.Init

Append / Delete / Done

Declaration
```
procedure Append(P : DoubleNodePtr);
```

Purpose
Add element to end of doubly linked list.

Description
Used just like SingleList.Append. For the doubly linked list, forward and backward links are forged for each element. DoubleList doesn't need to search to find the end of its list either, since it maintains a Tail field internally.

See Also
Insert
Place
SingleList.Append

Declaration
```
procedure Delete(P : DoubleNodePtr);
```

Purpose
Delete existing element in list, disposing of its contents.

Description
Used just like SingleList.Delete. Unlike SingleList, the DoubleList need not search to find the specified element P (because the node already specifies both its forward and backward links). The caller must therefore ensure that the element is already part of the list or Delete will corrupt memory while it attempts to unlink the element.

See Also
Remove
SingleList.Delete

Declaration
```
destructor Done; virtual;
```

Purpose
Destroy list.

Description
Disposes of all items on the list and then destroys the list itself.

See Also
SingleList.Done

Declaration
```
function Head : DoubleNodePtr;
```

Purpose
Return pointer to head of list.

Description
Works identically to SingleList.Head.

See Also
SingleList.Head Tail

Declaration
```
constructor Init;
```

Purpose
Initialize an empty list of items.

Description
Sets the head and tail pointers of the list to nil and the item count to zero.

See Also
Done SingleList.Init

Declaration
```
procedure Insert(P : DoubleNodePtr);
```

Purpose
Insert element at start of list.

Description
Used just like SingleList.Insert.

See Also
Append SingleList.Insert
PlaceBefore

Load / Next / Place

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a list from a stream.

Description

Used just like SingleList.Load. When using the IdStream.Get or IdStream.GetPtr methods for polymorphic loads, note that a DoubleList has its own object type code (otDoubleList) and version code (veDoubleList). The stream registration procedure for a DoubleList is DoubleListStream.

See Also

SingleList.Load Store

Declaration

```
function Next(P : DoubleNodePtr) : DoubleNodePtr; virtual;
```

Purpose

Returns a pointer to the next node.

Description

Used just like SingleList.Next. DoubleList.Next is a virtual method because it must be overridden when used in the CircularList object type.

See Also

Prev SingleList.Next

Declaration

```
procedure Place(P, L : DoubleNodePtr);
```

Purpose

Place element P into the list after existing element L.

Description

Used just like SingleList.Place.

See Also

PlaceBefore SingleList.Place

Declaration

```
procedure PlaceBefore(P, L : DoubleNodePtr);
```

Purpose

Place element P into the list before existing element L.

Description

Used just like SingleList.PlaceBefore. Unlike SingleList, the DoubleList need not search to find the existing element L (because a link to the previous element is stored within the node itself). The caller must therefore ensure that the element is already part of the list or PlaceBefore will corrupt memory while it attempts to link in the new element.

See Also

Place SingleList.PlaceBefore

Declaration

```
function Prev(P : DoubleNodePtr) : DoubleNodePtr; virtual;
```

Purpose

Returns a pointer to the previous node.

Description

Assumes that P is a valid node already in the list. If not, Prev will return garbage. Prev is a virtual method because it must be overridden when used in the CircularList object type.

Note that the function result must usually be cast to the type of the derived list node.

Example

```
NamePtr := NameDataPtr(NameList.Tail);
while NamePtr <> nil do begin
  NamePtr^.Display;
  NamePtr := NameDataPtr(NameList.Prev(NamePtr));
end;
```

Retrieve the pointer at the tail of the list. Then, using the Prev method, "walk" the linked list backwards, calling each object's Display method.

See Also

Next

Declaration

```
procedure Remove(P : DoubleNodePtr);
```

Purpose

Remove existing element from list without disposing of it.

Description

Used just like SingleList.Remove. Unlike SingleList, the DoubleList need not search to find the element P. The caller must therefore ensure that the element is already part of the list or Remove will corrupt memory while it attempts to unlink the element.

See Also

Delete SingleList.Remove

Size / Store / Tail

Declaration
```
function Size : Word;
```

Purpose
Return the size of the current list.

Description
The size is the number of elements in the list. This method works just like SingleList.Size.

See Also
SingleList.Size

Declaration
```
procedure Store(var S : IdStream);
```

Purpose
Store a list in a stream.

Description
Used just like SingleList.Store. When using the IdStream.Put or IdStream.PutPtr methods for polymorphic stores, note that a DoubleList has its own object type code (otDoubleList) and version code (veDoubleList). The stream registration procedure for a DoubleList is DoubleListStream.

See Also
Load SingleList.Store

Declaration
```
function Tail : DoubleNodePtr;
```

Purpose
Return pointer to tail of list.

Description
Reads an object data field that contains a pointer to the last element in the list. Returns nil if the list is empty.

Example
```
NamePtr := NameDataPtr(NameList.Tail);
NamePtr^.Display;
```

Retrieves the pointer at the tail of the NameList linked list (this would be the last pointer Appended) and calls that object's Display method. Since Tail returns a pointer of DoubleNodePtr type, we must typecast this to a type of NameDataPtr.

See Also
Head

Declaration
```
function Next(P : DoubleNodePtr) : DoubleNodePtr; virtual;
```

Purpose
Returns a pointer to the next node (wraps at end).

Description
Works just like DoubleList.Next with one exception. DoubleList.Next returns nil when its parameter is the last node on the list; CircularList.Next returns the first node in the list.

Examples
```
SavePtr := NamePtr;
repeat
  NamePtr^.Display;
  NamePtr := NameDataPtr(NameList.Next(NamePtr));
until NamePtr = SavePtr;
```

Notes the current pointer, then "walks" through the linked list until it circles back to the same pointer.

```
repeat
  NamePtr^.Display;
  NamePtr := NameDataPtr(NameList.Next(NamePtr));
until NamePtr = nil;
```

Incorrect! This is how a doubly-linked list works. Circular lists do not have any nil pointers. This repeat loop will continue forever.

Declaration
```
function Prev(P : DoubleNodePtr) : DoubleNodePtr; virtual;
```

Purpose
Returns a pointer to the previous node (wraps at begin).

Description
Works just like DoubleList.Prev with one exception. DoubleList.Prev returns nil when its parameter is the first node on the list; CircularList.Prev returns the last node in the list.

Declaration
```
procedure Store(var S : IdStream);
```

Purpose
Store a list in a stream.

Description
Works just like DoubleList.Store, except that it uses the list traversal method described under CircularList.Next to avoid looping forever.

CircularList uses the same Load method as DoubleList since the two lists are created the same way.

The stream registration procedure for a CircularList is CircularListStream.

See Also
DoubleList.Store DoubleList.Load

4. Bit Sets

The BitSet is a simple data structure that trades off speed to reduce memory usage. It performs the same function as does an array of Boolean: each element in an array can be marked either True or False. **Object Professional** uses the BitSet to mark selected items in multiple choice PickLists. Local instances of BitSets are used in other places as well: to find the next available window number in OPWINDOW and to indicate active fonts in OPFORM.

The BitSet uses each of the 8 bits of a byte to store the On/Off state of one element. It encapsulates the appropriate bit shifting and masking operations so that the application doesn't need to be aware of them. Methods defined by the BitSet allow setting, clearing, toggling, and testing particular bits, counting the number of set bits, and scanning rapidly from one set bit to the next.

Although the BitSet works much like Turbo Pascal's built-in sets, the BitSet's advantage is that it can store more than 256 elements. Its main disadvantage is that it doesn't have built-in compiler support for operations like initializing the set, creating set unions, and so on.

This section describes two different objects implementating the BitSet. The first object, called BitSet, limits the total elements in the set to 65534. The second, called LargeBitSet, allows up to 8 million bits in the set. As you might imagine, the second implementation is more complicated than the first and should be used only when absolutely required. Usage of both objects is almost identical, however, so switching between the two is easy.

The LargeBitSet object is so similar to the BitSet that we've saved space by not separately listing the methods that are used just like those in the BitSet. Only those methods that are new or significantly different get a separate entry. Otherwise, you can refer to the BitSet methods for information about the LargeBitSet object.

The LargeBitSet object has one additional advantage over the BitSet: LargeBitSets can be resized dynamically. This makes them good candidates for operations where the number of elements is potentially large but not known in advance.

The following program example demonstrates some of the features of bit sets.

```
program BitSetExample;  {EXBITSET.PAS}
uses
  OpRoot;
var
  DivisibleBy9 : BitSet;
  I, Cnt : Integer;
begin
  {Create a BitSet for flagging 1000 items (this clears all bits)}
  if not DivisibleBy9.Init(1000) then begin
    Writeln('Failed to Init BitSet,  Status = ', InitStatus);
    Halt;
  end;

  {Loop through first 1000 integers, set bit if integer is divisible by 9}
  for I := 1 to 1000 do
    if (I mod 9) = 0 then
      DivisibleBy9.SetBit(I);

  {How many, total, are divisible by 9?}
  Writeln(DivisibleBy9.BitsSet, ' are divisible by 9');

  {How many, between 500 and 600, are divisible by 9?}
  Cnt := 0;
```

```
  for I := 500 to 600 do
    if DivisibleBy9.BitIsSet(I) then
      Inc(Cnt);
  WriteLn(Cnt, ' between 500 and 600 are divisible by 9');

  DivisibleBy9.Done;
end.
```

Initializing a BitSet allocates enough heap space to hold the requested number of bits. In the example, 1000 bits require 126 bytes of heap space. Calling BitSet.Init also clears all the bits. The example program sets only those bits whose index is divisible by 9. Many other operations similar to SetBit are available--ClearBit, ToggleBit, ControlBit, TestSetBit, and so on.

BitsSet returns the total number of bits set (in the range 1..MaxBit, the 0 bit is a special case). BitIsSet tests the state of one bit.

Finally, calling DivisibleBy9.Done deallocates the heap space used for the BitSet and shuts down the object.

Constants

```
MaxBitCount = 65534;
```

Maximum bits in a BitSet.

```
MaxLargeBitCount = 1024*1024*8;
```

Maximum bits in a LargeBitSet.

```
NoMoreBits = $FFFFFFFF;
```

Constant returned by FirstSet, LastSet, and other methods when no more bits match the specified criterion.

Types

```
BitSetPtr = ^Bitset;
BitSet =
  object(Root)
    ...
  end;

LargeBitSetPtr = ^LargeBitset;
LargeBitSet =
  object(BitSet)
    ...
  end;
```

Data types for managing bit sets. Note that LargeBitSet is derived from BitSet.

BitIsSet / BitsSet

Declaration

```
function BitIsSet(Element : LongInt) : Boolean; virtual;
```

Purpose

Return True if bit is set.

Description

The valid range for Element is from 0 to MaxBits (the maximum specified when the BitSet was constructed). For any other value BitIsSet always returns False.

You can test whether a bit is clear by calling BitIsSet and complementing the boolean result.

This method is virtual because it is overridden in the LargeBitSet object.

Example

```
Cnt := 0;
for I := 500 to 600 do
  if DivisibleBy9.BitIsSet(I) then Inc(Cnt);
WriteLn(Cnt,' are divisible by 9');
```

Assuming that each bit in DivisibleBy9 represents an integer that is exactly divisible by nine, this example counts the number of integers between 500 and 600 that are exactly divisible by nine.

See Also

FirstSet

Declaration

```
function BitsSet : LongInt; virtual;
```

Purpose

Return total number of bits set.

Description

Scans the entire bitset (from 1 to MaxBits) to count the number of bits that are set. For large bitsets this may be a time-consuming operation. To minimize the time, BitsSet calls NextSet, which uses an inline macro to scan as rapidly as possible from one set bit to the next.

The 0'th bit is not counted in BitsSet regardless of its state. The number of clear bits is given by MaxBits-BitsSet, again ignoring the state of the 0'th bit.

Example

```
Writeln(DivisibleBy9.BitsSet, ' are divisible by 9');
```

Assuming DivisibleBy9 has a bit set for each integer between 1 and 1000 that is evenly divisible by nine, this example reports the total number of qualifying integers.

Declaration
```
procedure ClearAll; virtual;
```

Purpose
Clear all bits.

Description
Resets the bitset to the state it had when it was created (no bits set).

Example
```
DivisibleBy9.ClearAll;
```
Clears all bits in the DivisibleBy9 bitset to zero.

See Also
SetAll

Declaration
```
procedure ClearBit(Element : LongInt); virtual;
```

Purpose
Clear bit.

Description
Clears the specified bit. The valid range for Element is from 0 to MaxBits (the maximum specified when the BitSet was constructed). For any other value ClearBit does nothing.

Example
```
DivisibleBy9.SetAll;
for I := 1 to 1000 do
  if (I mod 9) = 0 then
    DivisibleBy9.ClearBit(I);
```
Starts out with all bits set, then clears every bit that represents an integer that is evenly divisible by nine.

See Also
SetBit ToggleBit

ControlBit / Done / FirstClear

Declaration
```
procedure ControlBit(Element : LongInt; State : Boolean);
```

Purpose
Set or clear bit depending on State.

Description
This method internally calls either SetBit or ClearBit depending on the value of State. It does nothing for an invalid Element number.

Example
```
OptionBits.ControlBit(ThisOption, ThisOptionDefault);
```
Assuming OptionBits contains a set of program option flags, this example modifies the bit that represents ThisOption. If ThisOptionDefault is True, then ThisOption bit is set, otherwise it is cleared.

See Also
ToggleBit

Declaration
```
destructor Done; virtual;
```

Purpose
Deallocate data area for BitSet.

Description
Deallocates the heap space allocated for BitSet storage.

Example
```
DivisibleBy9.Done;
```
Frees the heap space required for the BitSet DivisibleBy9.

See Also
Init

Declaration
```
function FirstClear : LongInt;
```

Purpose
Return index of first clear bit, NoMoreBits if none.

Description
Internally calls the NextClear method, which uses an inline macro to find the first clear bit as rapidly as possible. FirstClear always ignores the 0'th bit; the lowest value it will ever return is 1.

NoMoreBits is a constant equaling $FFFFFFFF.

See Also
FirstSet LastClear

Declaration
```
function FirstSet : LongInt;
```

Purpose
Return index of first set bit, NoMoreBits if none.

Description
Internally calls NextSet, which uses an inline macro to find the first set bit as rapidly as possible. FirstSet always ignores the 0'th bit; the lowest value it will ever return is 1.

NoMoreBits is a constant equaling $FFFFFFFF.

Example
```
WriteLn('The first qualifying integer is ', DivisibleBy9.FirstSet);
```
Reports the position of the first set bit in the BitSet DivisibleBy9.

See Also
FirstClear LastSet

Declaration
```
constructor Init(Max : LongInt);
```

Purpose
Allocate space for elements.

Description
Init allocates ((Max+8) div 8) bytes of heap space (for the small BitSet). If there is insufficient space, the constructor will fail and an error code will be stored in the global variable InitStatus. If Max is larger than MaxBitCount, Init will also fail.

Valid elements to use in the BitSet range from 0 to Max. However, bit 0 is used as a sentry bit for "scanning" operations such as NextSet and FirstSet. Hence, the use of this bit is not recommended when these operations are required. If only SetBit, ClearBit, and BitIsSet will be called, bit 0 may be used like any other set element.

If the value of Max is 0, Init initializes its internal data values (all to zero), but allocates no heap space. This is sometimes useful when constructing objects that depend on the BitSet. It is safe to call BitSet.Done after this form of call to Init.

Examples
```
if not DivisibleBy9.Init(1000) then begin
  Writeln ('Failed to Init BitSet,  Status = ', InitStatus);
  Halt;
end;
```
Creates a BitSet called DivisibleBy9 that can store up to 1000 bits.
```
if not DivisibleBy9.Init(100000) then begin
  Writeln ('Failed to Init BitSet,  Status = ', InitStatus);
  Halt;
end;
```
Incorrect! The limit for a Bitset is 65534 bits. Use LargeBitSet for sets greater than this.

See Also
Done LargeBitSet.Init

LastClear / LastSet

Declaration

```
function LastClear : LongInt;
```

Purpose

Return index of last clear bit, NoMoreBits if none.

Description

Internally calls the PrevClear method, which uses an inline macro to find the previous clear bit as rapidly as possible. LastClear always ignores the 0'th bit; the lowest value it will ever return is 1.

NoMoreBits is a constant equaling $FFFFFFFF.

See Also

FirstClear

Declaration

```
function LastSet : LongInt;
```

Purpose

Return index of last set bit, NoMoreBits if none.

Description

Internally calls the PrevSet method, which uses an inline macro to find the previous set bit as rapidly as possible. LastSet always ignores the 0'th bit; the lowest value it will ever return is 1.

NoMoreBits is a constant equaling $FFFFFFFF.

See Also

FirstSet

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a BitSet from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to BitSet.Store. Load allocates heap space for the BitSet and then reads an image of the BitSet from the stream.

If Load fails, the constructor will return False and an error code will be stored in the global variable InitStatus.

The stream registration procedure for a BitSet is BitSetStream.

Example

```
uses
  OpRoot;
var
  DivisibleBy9 : BitSet;
  S : BufIdStream;
  I, Cnt : Integer;
begin
  {Read a BitSet from a stream}
  S.Init('BITSET.STM', SOpen, 1024);
  if not DivisibleBy9.Load(S) then begin
    WriteLn('Load error, ', InitStatus);
    Halt;
  end;
  S.Done;

  {How many, between 500 and 600, are divisible by 9?}
  Cnt := 0;
  for I := 500 to 600 do
    if DivisibleBy9.BitIsSet(I) then
      Inc(Cnt);
  WriteLn(Cnt,' are divisible by 9');

  {Dispose of the BitSet}
  DivisibleBy9.Done;
end.
```

This example program loads the DivisibleBy9 BitSet from a stream created in the Store example.

An effect similar to calling Load may be achieved with the IdStream.Get method. See PointerStack.Load for further information.

See Also

IdStream.Get Store

MaxBits / NextClear

Declaration

```
function MaxBits : LongInt;
```

Purpose

Return capacity of BitSet.

Description

Returns the same maximum value that was passed to the BitSet's constructor. The valid range of bits is 0..MaxBits. However, the 0'th bit is reserved as a sentry position for operations that scan from bit to bit (NextSet, PrevSet, etc.). Use of this bit should be avoided in most cases.

Example

```
WriteLn('The capacity of this BitSet is ', DivisibleBy9.MaxBits);
```

Reports the bit capacity of DivisibleBy9.

See Also

BitSet Init

Declaration

```
function NextClear(Element : LongInt) : LongInt; virtual;
```

Purpose

Return index of next clear element, NoMoreBits if none.

Description

Uses an inline macro to scan to the next clear bit as rapidly as possible.

See Also

NextSet PrevClear

Declaration

```
function NextSet(Element : LongInt) : LongInt; virtual;
```

Purpose

Return index of next set element, NoMoreBits if none.

Description

Uses an inline macro to scan to the next set bit as rapidly as possible. Element should be a valid index in the BitSet; NextSet starts searching at a position 1 higher than Element.

Be careful when checking to see whether NextSet has returned NoMoreBits as a result. NoMoreBits is a LongInt constant; if the result of NextSet is assigned to a Word variable the following typecast is required:

```
WordVar := NextSet(Element);
if WordVar = Word(NoMoreBits) then
  {no more set bits} ;
```

Examples

```
I := DivisibleBy9.NextSet(500); {I is a LongInt variable}
if I = NoMoreBits then
  WriteLn('No bits set after 500')
else
  WriteLn(I, ' is the next bit set after 500');
```

Reports the next bit set after bit number 500. If no more set bits exist, NoMoreBits ($FFFFFFFF) is returned.

```
I := 0; {I is a LongInt variable}
repeat
  I := DivisibleBy9.NextSet(I);
  if I <> NoMoreBits then
    WriteLn(I, ' is set');
until I = NoMoreBits;
```

Scans all set bits in DivisibleBy9.

See Also

NextClear PrevSet

Declaration

```
function PrevClear(Element : LongInt) : LongInt; virtual;
```

Purpose

Return index of previous clear element, NoMoreBits if none.

Description

Uses an inline macro to scan to the previous clear bit as rapidly as possible.

See Also

PrevSet NextClear

PrevSet / SetAll

Declaration

```
function PrevSet(Element : LongInt) : LongInt; virtual;
```

Purpose

Return index of previous set element, NoMoreBits if none.

Description

Uses an inline macro to scan to the previous set bit as rapidly as possible. With one exception, Element should always be a valid index in the BitSet. PrevSet starts searching at a position 1 lower than Element. It is OK to pass a value of MaxBits+1 as the argument in order to find the last set bit.

Be careful when checking to see whether PrevSet has returned NoMoreBits as a result. NoMoreBits is a LongInt constant; if the result of PrevSet is assigned to a Word variable the following typecast is required:

```
WordVar := PrevSet(Element);
if WordVar = Word(NoMoreBits) then
  {no more set bits} ;
```

Examples

```
I := DivisibleBy9.PrevSet(500); {I is a LongInt variable}
if I = NoMoreBits then
  WriteLn('No bits set before 500')
else
  WriteLn(I,' is the next bit set before 500');
```

Reports the next bit set before bit number 500. If no more set bits exist, NoMoreBits ($FFFFFFFF) is returned.

```
I := DivisibleBy9.MaxBits+1; {I is a LongInt variable}
repeat
  I := DivisibleBy9.PrevSet(I);
  if I <> NoMoreBits then
    WriteLn(I, ' is set');
until I = NoMoreBits;
```

Scans all set bits in DivisibleBy9, in reverse order.

See Also

PrevClear NextSet

Declaration

```
procedure SetAll; virtual;
```

Purpose

Set all bits.

Description

Fills the entire BitSet with $FF bytes, thus setting all bits to 1.

Example

```
DivisibleBy9.Init(1000);
DivisibleBy9.SetAll;
```

Starts out with all of the bits set (to 1).

See Also

ClearAll

Declaration

```
procedure SetBit(Element : LongInt); virtual;
```

Purpose

Set bit.

Description

Sets the specified bit. The valid range for Element is from 0 to MaxBits (the maximum specified when the BitSet was constructed). For any other value SetBit does nothing.

Example

```
for I := 1 to 1000 do
  if (I mod 9) = 0 then
    DivisibleBy9.SetBit(I);
```

Loops through the first 1000 integers, setting a bit flag whenever a particular integer is divisble by nine.

See Also

ClearBit
ControlBit
ToggleBit

Store

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a BitSet to a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the BitSet to the stream S which may be read by a later call to Load.

To check for errors that may have occurred during the Store, call the stream's GetStatus function as shown in the example below.

The stream registration procedure for a BitSet is BitSetStream.

Example

```
uses OpRoot;
var
  DivisibleBy9 : BitSet;
  S : BufIdStream;
  I, Cnt : Integer;
  Status : Word;
begin
  {Create a BitSet for flagging 1000 items (this clears all bits)}
  if not DivisibleBy9.Init(1000) then begin
    Writeln ('Failed to Init BitSet,  Status = ', InitStatus);
    Halt;
  end;
  {Loop through first 1000 integers, set bit if integer is divisible by 9}
  for I := 1 to 1000 do
    if (I mod 9) = 0 then
      DivisibleBy9.SetBit(I);
  {Write this BitSet to a stream}
  S.Init('BITSET.STM', SCreate, 1024);
  DivisibleBy9.Store(S);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error, ', Status);
    Halt;
  end;
  S.Done;
  DivisibleBy9.Done;
end.
```

This example program creates the DivisibleBy9 BitSet, sets some bits, then writes the BitSet to a stream. See the Load example for a program that will load and use this stream.

An effect similar to calling Store may be achieved with the IdStream.Put method. IdStream.Put is often desirable because it is polymorphic; that is, it automatically determines the appropriate Store method to call based on the stream's data. When a stream is written with Put, the same stream must be read with Get. Get can then determine automatically which Load constructor to call. To use Put, the Store call in the example above may be replaced with

```
S.RegisterHier(BitSetStream);
S.Put(DivisibleBy9);
```

See Also

IdStream.Put
IdStream.RegisterType
Load

Declaration

```
function TestClearBit(Element : LongInt) : Boolean;
```

Purpose

Return current state of bit and clear it.

Description

This method combines the functions of BitIsSet and ClearBit to provide slightly better efficiency when both are required.

Example

```
if OptionBits.TestClearBit(ThisOption) then
  WriteLn('ThisOption stopped')
else
  WriteLn('ThisOption was already off');
```

Assuming OptionBits contains a set of program option flags, this example turns off ThisOption and reports its previous state.

Declaration

```
function TestSetBit(Element : LongInt) : Boolean;
```

Purpose

Return current state of bit and set it.

Description

This method combines the functions of BitIsSet and SetBit to provide slightly better efficiency when both are required.

Example

```
if OptionBits.TestSetBit(ThisOption) then
  WriteLn('ThisOption was already on')
else
  WriteLn('ThisOption started');
```

Assuming OptionBits contains a set of program option flags, this example turns on ThisOption and reports its previous state.

Declaration

```
procedure ToggleBit(Element : LongInt);
```

Purpose

Toggle bit.

Description

Reverses the setting of the specified element.

Example

```
OptionBits.ToggleBit(ThisOption);
```

Assuming OptionBits contains a set of program option flags, this example toggles the state of the bit that represents ThisOption. If ThisOption was set, this clears it. If it was clear, this sets it.

See Also

ClearBit SetBit

Error / GetStatus

Declaration

```
procedure Error(Code : Word); virtual;
```

Purpose

Report an error.

Description

Unlike BitSet, the LargeBitSet needs an internal status code because it may attempt to allocate heap space during SetBit operations. When an error occurs, the LargeBitSet methods call the Error routine to report the error. By default, the Error routine simply stores Code in an internal status variable. Types derived from LargeBitSet may override the Error method to implement a different error reporting strategy.

Declaration

```
function GetStatus : Word;
```

Purpose

Return latest status code and reset internal status to zero.

Description

Unlike BitSet, the LargeBitSet needs an internal status code because it may attempt to allocate heap space during SetBit operations. When the caller of SetBit specifies an element number larger than the previous MaxBits limit, SetBit calls LargeBitSet.Resize to expand the BitSet to encompass the new element. Resize can report either of two errors, out of memory or invalid parameter. The program can call GetStatus after such a call to detect errors.

If calls to SetBit will never cause the BitSet to expand, there is no need to call GetStatus.

Example

```
{...SetBit or Resize...}
Status := DivisibleBy9.GetStatus;
if Status <> 0 then begin
  WriteLn('Operation failed, Status = ', Status);
  Halt;
end;
```

Call GetStatus to check the status of the last SetBit or Resize operation. Since referencing GetStatus clears the status code, move the status code into a temporary variable to use it more than once.

See Also

Resize

Declaration

```
constructor Init(Max : LongInt);
```

Purpose

Allocate space for elements and clear all the bits.

Description

Init allocates heap space in 512 byte pages, each capable of holding 4096 bits. It also allocates an array of pointers to manage the pages. For example, if you were initializing a BitSet with 100,000 elements, Init would allocate

```
25 pages, each of 512 bytes
1 page array of 4*25 bytes
```

for a total of 12900 bytes. In the best of all worlds the BitSet would require 12501 bytes, so the overhead of managing the LargeBitSet is only about 3% in this case. The largest acceptable value for Max is MaxLargeBitCount, about 8 million bits.

The LargeBitSet can be resized dynamically, so it can deal with situations where the upper limit on bits isn't known in advance. The SetBit and Resize methods can be used after initializing the BitSet in order to increase the upper limit.

If the value of Max is 0, Init initializes its internal data values (all to zero), but allocates no heap space. This is sometimes useful when constructing objects that depend on the LargeBitSet. It is safe to call LargeBitSet.Done after this form of call to Init.

Example

```
if not DivisibleBy9.Init(100000) then begin
  Writeln ('Failed to make BitSet,  Status = ', InitStatus);
  Halt;
end;
```

Creates a BitSet of 100,000 bits (absolute maximum is 8,388,608 bits).

See Also

Resize

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a large BitSet from a stream.

Description

Used just like BitSet.Load. When using the IdStream.Get methodology, note that LargeBitSet has its own object type (otLargeBitSet) and version code (veLargeBitSet) which should be used in registering the type with the stream. The stream registration procedure for a LargeBitSet is LargeBitSetStream.

See Also

IdStream.Get

IdStream.RegisterType

Resize / Store

Declaration

```
function Resize(Max : LongInt) : Boolean;
```

Purpose

Expand or shrink large BitSet to hold Max bits.

Description

LargeBitSets can be dynamically resized with little overhead. This makes them ideal for applications where the possible number of elements is large but not known in advance.

Resize allocates additional 512 byte pages for the BitSet if the new maximum number of elements exceeds the old. The new pages are filled with zeros and the old pages retain their previous values. If the new Max is smaller than the old, pages are deallocated.

Resize returns False if there was insufficient memory available to allocate new pages, or if the value of Max is invalid (0 or greater than MaxLargeBitCount). An exact error code can be retrieved by calling GetStatus.

Example

```
if not DivisibleBy9.Init(100000) then begin
  Writeln ('Failed to make BitSet,  Status = ', InitStatus);
  Halt;
end;

if not DivisibleBy9.Resize(200000) then begin
  WriteLn('Failed to Resize BitSet, Status = ', DivisibleBy9.GetStatus);
  Halt;
end;

if not DivisibleBy9.Resize(50000) then begin
  WriteLn('Failed to Resize BitSet, Status = ', DivisibleBy9.GetStatus);
  Halt;
end;
```

First initialize a large BitSet to hold 100,000 bits. Later expand this BitSet to hold 200,000 bits. The boolean function Resize returns False if it couldn't resize the array. In this example, that could only be due to an "out of memory" error. Finally, shrink the BitSet to hold only 50,000 bits. This operation should never fail.

See Also

GetStatus

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a BitSet to a stream.

Description

Used just like BitSet.Store. When using the IdStream.Put methodology, note that LargeBitSet has its own object type (otLargeBitSet) and version code (veLargeBitSet) which should be used in registering the type with the stream. The stream registration procedure for a LargeBitSet is LargeBitSetStream.

See Also

IdStream.Put

5. String Management

This section collects information about three related objects, all of which are useful for managing strings.

The StringDict implements a "string dictionary." What this does is quite similar to the dictionary on your bookshelf. You can look up an item by name, and the dictionary will either return some information about the item or else it will say that the string isn't available. Extending the metaphor, you can also add your own strings to the StringDict or remove existing entries.

The StringDict reserves only 4 bytes of storage to hold auxiliary information about each string. This is plenty if you consider that the 4 bytes can be used as a pointer to a larger data structure you supply. In many cases, however, all you'll want the StringDict to return is a parser token--a small unique integer that indicates the meaning of the string to some kind of parser.

StringDict uses hashing technology to quickly find a given string in its internal table. Hashing is one of the fastest methods available for searching when all the items will fit into memory. It definitely outperforms linear searching of an array, and almost always beats out binary searching of a sorted array or traversal of a binary tree.

A StringSet is an object derived from a StringDict. The StringSet's restricted subset of the StringDict's capabilities allow you only to detect whether a particular string is in the dictionary or not; no additional information is returned. As a result the StringSet is slightly more efficient than the StringDict. **Object Professional** uses a StringSet in the DirList object to decide whether to accept or reject files with particular extensions.

A StringArray provides a different kind of capability than the other two objects. It's often convenient to use a normal Pascal array of strings in a program, perhaps to store a list of help messages or to provide the items in a pick list. The disadvantage of a normal Pascal array is that each element of the array must be as large as the longest string. That potentially leads to lots of wasted space.

The StringArray packs each string end-to-end so that the array is no larger than it needs to be. It also maintains an index to the array so that it can quickly find each string. In most cases, the overhead of the index (2 bytes per string) is much less than the space saved by packing the strings.

The StringArray provides operations to read and write text files of strings, add new strings, and return existing strings. It does not allow deletion or modification of existing strings.

The following example demonstrates typical usage of a StringDict:

```
program StringDictionaryExample;  {EXSDICT.PAS}
uses
  OpRoot;
type
  NameDataPtr = ^NameData;
  NameData = record
    Age : Byte;
    Addr : String;
  end;
var
  NamePtr : NameDataPtr;
  NameDict : StringDict;
  Status : Word;
```

```
begin
  {Initialize a StringDict}
  if not NameDict.Init then begin
    Writeln('Failed to make StringDict,  Status = ', InitStatus);
    Halt;
  end;

  {Create and add some names to the StringDict}
  New(NamePtr);
  with NamePtr^ do begin
    Age := 20;
    Addr := '123 Vintage Lane';
  end;
  NameDict.Add('Nessus', LongInt(NamePtr));

  New(NamePtr);
  with NamePtr^ do begin
    Age := 46;
    Addr := 'Long House';
  end;
  NameDict.Add('Beowulf', LongInt(NamePtr));

  {Check status after Adds}
  Status := NameDict.GetStatus;
  if Status <> 0 then begin
    WriteLn('Failed to add Names, Status = ', Status);
    Halt;
  end;

  {Search for a particular name}
  if NameDict.Member('Nessus', LongInt(NamePtr)) then begin
    WriteLn('Found');
    with NamePtr^ do begin
      WriteLn('Age ', Age);
      WriteLn('Address ', Addr);
    end;
  end else
    Writeln('Not found');

  {Retrieve and dispose the NamePtrs}
  if NameDict.Member('Nessus', LongInt(NamePtr)) then
    Dispose(NamePtr);
  if NameDict.Member('Beowulf', LongInt(NamePtr)) then
    Dispose(NamePtr);
  NameDict.Done;
end.
```

When NameDict is initialized it creates a small initial hash table capable of holding at most 16 entries. The object will automatically expand this hash table when it is in danger of overflowing.

The example dictionary uses a person's name as the dictionary lookup, and stores a pointer to more details about the person. We allocate and initialize two of these description records and add two names to the dictionary. Note that the pointer passed to the Add method must be typecast to a LongInt to make it acceptable to the routine.

Each call to Add requires the StringDict to allocate heap space to store the dictionary name. Also, if enough strings are added, the hash table must be expanded, requiring even more heap space. Therefore it is a good idea to follow calls to Add with a check of GetStatus.

The Member function returns True if the specified string is in the dictionary. It also returns the data value originally stored with the string. In this case, that data value was a pointer to a more detailed description, which we can dereference and display.

Further calls to Member in this example program serve to get the NameData pointers so that we can dispose of their heap space. Finally, a call to NameDict.Done disposes of the dictionary itself.

The StringSet is so similar to the StringDict that we won't show an example for it here. The primary differences are that the Add and Member functions don't take a second parameter.

The StringArray does deserve an example of its own.

```
program StringArrayExample;  {EXSARRAY.PAS}
uses
  OpRoot;
var
  Messages : StringArray;
  Failed : Boolean;
begin
  {Create a StringArray (estimate 10 messages at 20 characters each)}
  if not Messages.Init(10, 200) then begin
    WriteLn('Failed to Create StringArray,  Status = ', InitStatus);
    Halt;
  end;

  {Add some messages}
  Failed := False;
  if Messages.AddString('Out of Memory') = 0 then
    Failed := True;
  if Messages.AddString('Parameter out of range') = 0 then
    Failed := True;
  if Messages.AddString('Busy') = 0 then
    Failed := True;

  {Did any of the Adds fail?}
  if Failed then begin
    Writeln('Out of Memory');
    Halt;
  end;

  {Display a Message}
  WriteLn(Messages.GetString(2));

  {Dispose of the StringArray}
  Messages.Done;
end.
```

The constructor call for a StringArray requires us to estimate how many strings will be in the array and how many bytes those strings will use. An alternate constructor allows reading the strings from a text file without providing any estimates, so for large applications there is an automatic way to determine these numbers. AddString will also automatically expand the tables as needed, so the initial limits don't pose any real problem.

The AddString function appends the string to the packed array and returns the string index number for the new string. If AddString returns 0, the StringArray is full and cannot be expanded. Often you'll want to save the index number returned by AddString for reference later in a program. Here we don't care.

The GetString call returns the string with the specified index. And of course Done disposes of the StringArray's heap space and destroys the object itself.

Constants

```
InitPoolSize = 16;
```

The initial hash table size for a StringDict. Must always be a power of two.

```
paGrowIndex : Word = 100;
```

When a StringArray is forced to expand its index table, it increases the available number of entries by paGrowIndex. This value can be changed at any point prior to the time when the StringArray expands.

```
paGrowInfo  : Word = 1000;
```

Number of bytes by which a StringArray expands its packed array when forced to grow. The index and packed array are expanded independently as necessary.

Types

```
StringDictPtr = ^StringDict;
StringDict =
  object(Root)
    ...
  end;
StringSetPtr = ^StringSet;
StringSet =
  object(StringDict)
    ...
  end;
StringArrayPtr = ^StringArray;
StringArray =
  object(Root)
    ...
  end;
```

The data types defining the string management objects described in this section.

Declaration

```
procedure Add(S : String; Value : LongInt);
```

Purpose

Add string and associated value to dictionary.

Description

If the string is already in the dictionary (by comparing to existing entries using the Hash and Equal functions), Add generates a warning (epWarning+ecDupString) and exits without changing the dictionary. Otherwise, it allocates heap space just large enough to store the string and its LongInt data value, and adds an entry to its internal hash table.

StringDict reserves 4 bytes of each dictionary entry for user data storage. It declares this data area as a LongInt to make it as easy as possible to store values without typecasting. Another common type for the data value is a pointer, which fits into the same space as a LongInt. When passing a pointer to Add, it must be typecast to a LongInt to circumvent Pascal's strong type checking. Whenever the dictionary must store more than 4 bytes of information about each string, it should instead store a pointer to a record, where the record stores the actual information.

If the hash table is more than 75% full after the string has been added, Add expands the hash table by a factor of two. This involves allocating a new hash table, transferring all the old table entries to the new table, and deallocating the old one.

Because Add may generate warnings and errors, it should usually be followed by a call to GetStatus. If an error or warning is pending at the time Add is called, Add exits immediately without doing anything. Hence, it's usually adequate to have just one GetStatus check at the end of a series of Add's.

Note that, by default, the StringDict ignores case in deciding whether two strings are the same. You can change this behavior by overriding the Hash and Equal methods.

Examples

```
var
  ParseDict : StringDict;
begin
  ParseDict.Init;
  ParseDict.Add('BEGIN', 1);
  ParseDict.Add('END',   2);
  ParseDict.Add('IF',    3);
  ParseDict.Add('THEN',  4);
  if ParseDict.GetStatus <> 0 then begin
    WriteLn('Error adding strings');
    Halt;
  end;
end.
```

Initializes a dictionary that might be useful for writing a small computer language parser.

```
type
  NameDataPtr = ^NameData;
  NameData = record
    Age : Byte;
    Addr : String;
  end;
var
  NamePtr : NameDataPtr;
  NameDict : StringDict;
...
  New(NamePtr);
  NameDict.Add('Nessus', LongInt(NamePtr));
  New(NamePtr);
  NameDict.Add('Beowulf', LongInt(NamePtr));
```

```
Status := NameDict.GetStatus;
if Status <> 0 then begin
  WriteLn('Failed to add Names, Status = ', Status);
  Halt;
end;
```

Allocates NameData records on the heap (though it doesn't put any information in them). It then associates each of these records with a string and adds the string and pointer to the NameDict string dictionary.

We chose to check for errors only after we added both strings. You may want to check for errors after each add. The Add method can return "out of memory" errors and "dup string" errors. In this example, if GetStatus reported a duplicate string, we wouldn't know which of our two strings was the duplicate.

See Also

Equal
Hash
Remove
Update

Declaration
```
procedure Clear;
```

Purpose
Remove all strings from dictionary.

Description
Deallocates string space and clears the internal hash tables. If dictionary elements are associated with separate dynamic storage (as in the NameDict example), be sure to deallocate the separate storage before clearing the dictionary.

The string dictionary still remains available for use after this call.

Examples
```
NameDict.Clear;
```

Removes all strings from the NameDict dictionary.

```
if NameDict.Member('ABC',LongInt(NamePtr)) then
  Dispose(NamePtr);
NameDict.Clear;
```

Disposes of associated heap data before clearing the string dictionary.

Declaration
```
destructor Done; virtual;
```

Purpose
Deallocate string dictionary.

Description
First clears the dictionary, then deallocates the internal hash table and destroys the object.

Example
```
NameDict.Done;
```

Deallocates the heap space used by the string dictionary. (Doesn't deallocate application-specific heap space used for each element.)

Declaration
```
function Equal(var S, T : String) : Boolean; virtual;
```

Purpose
Return True if the two strings are considered equal.

Description
This method is called by other StringDict methods whenever two strings must be compared for equality. It ignores case in making the comparison. If you have an application that requires case sensitivity, you must override both this method and the Hash function.

Example
```
function CaseStringDict.Equal(var S, T : String) : Boolean;
begin
  Equal := (S = T);
end;
```

Shows the Equal method required for a case-sensitive string dictionary derived from StringDict.

See Also
Hash

Error / GetStatus

Declaration

```
procedure Error(Code : Word); virtual;
```

Purpose

Report an error.

Description

Applications do not call this routine directly. When a StringDict method detects an error, it calls the Error method to report it. The default implementation of this method simply assigns the error code to an internal status variable. An object derived from StringDict may override the Error method to provide different error handling behavior.

Declaration

```
function GetStatus : Word;
```

Purpose

Return status code and reset internal result to zero.

Description

Each StringDict maintains an internal variable that stores the current status of the dictionary. GetStatus provides access to this variable. Like IoResult, GetStatus clears the internal status variable after it returns the current value.

When no error has occurred, GetStatus returns 0. Otherwise it returns an error code as specified in Appendix D. Expected error values include

```
epFatal+ecOutOfMemory
epWarning+ecDupString
```

Example

```
ParseDict.Add('BEGIN', 1);
Status := ParseDict.GetStatus;
if Status <> 0 then begin
  WriteLn('Add failed, Status = ', Status);
  Halt;
end;
```

Add is the only StringDict method that might fail (other than Init). Since referencing the function GetStatus clears the current value, this example saves that value in our variable, then displays that variable.

Declaration

```
function Hash(var S : String) : Word; virtual;
```

Purpose

Return a hash code for the specified string.

Description

This method is called by other StringDict methods whenever they are locating the first probe point in the internal hash table. It ignores case in computing the hash value. If you have an application that requires case sensitivity, you must override both this method and the Equal function.

Hash must return a value between 0 and (sdSize-1), inclusive, where sdSize is the current number of elements in the hash table.

For certain collections of strings, an alternate hash function may improve the performance of the string dictionary. Finding an optimized hash function is a difficult job, however.

Example

```
function CaseStringDict.Hash(var S : String) : Word;
var I, Sum : Word;
begin
  Sum := 0;
  for I := 0 to Length(S) do
    Sum := (Sum shl 1)+Byte(S[I]);
  Hash := Sum and (sdSize-1);
end;
```

Shows a Hash method suitable for a case-sensitive string dictionary derived from StringDict.

See Also

Equal

Declaration

```
constructor Init;
```

Purpose

Allocate hash table for a string dictionary.

Description

This allocates an initial hash table with space for InitPoolSize entries. (InitPoolSize is a constant with a value of 16.) Each entry uses 4 bytes, so Init will typically allocate 64 bytes of heap space. The hash table is cleared so that the StringDict is initially empty.

If the constructor fails, it returns a boolean result of False or a pointer result of nil, depending on whether the StringDict is defined as a static or dynamic instance. Error status is returned in the OPROOT global variable InitStatus.

The StringDict object automatically expands itself whenever it becomes more than 75% full. Due to implementation decisions made for the sake of efficiency, the absolute maximum number of strings in one string dictionary is limited to 8000.

Example

```
if not NameDict.Init then begin
  Writeln('Failed to make StringDict. Status = ', InitStatus); Halt;
end;
```

Initializes a string dictionary named NameDict.

See Also

Done

Load

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a string dictionary from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to StringDict.Store. Load allocates heap space for the hash table, then reads a series of strings and data values from the stream and adds them to the dictionary.

If Load fails, the constructor will return False and an error code will be stored in the global variable InitStatus.

The stream registration procedure for a StringDict is StringDictStream.

Example

```
uses
  OpRoot;
var
  NameDict : StringDict;
  NDS : BufIdStream;
  Data : LongInt;
begin
  {Read a StringDict from a stream}
  NDS.Init('STRDICT.STM', SOpen, 1024);
  if not NameDict.Load(NDS) then begin
    WriteLn('Load error, ', InitStatus);
    Halt;
  end;
  NDS.Done;

  {Lookup the strings and display their data}
  if NameDict.Member('Nessus', Data) then
    WriteLn(Data);
  if NameDict.Member('Beowulf Shaeffer', Data) then
    WriteLn(Data);
  NameDict.Done;
end.
```

Loads a string dictionary from the stream created in the Store example. That stream contains two strings and and their "data" (just LongInts in this case). We instantiate NameDict from the stream and retrieve the two strings and display their data.

An effect similar to calling Load may be achieved with the IdStream.Get method. See PointerStack.Load for further information.

See Also

IdStream.Get Store

Declaration

```
function Member(S : String; var Value : LongInt) : Boolean;
```

Purpose

Return True and associated data value if S is element of dictionary.

Description

The Member function computes the hash of the string S and then probes the internal hash table to find the string. Typically this will require only a few string comparisons even for very large dictionaries.

If a matching string is found (ignoring case), Member returns True and initializes the Value result to the user value stored when the string was first added. Note that Value is of type LongInt: even if the actual data is a byte, the parameter passed must be a LongInt. A pointer variable may also be used with appropriate typecasting.

Example

```
if NameDict.Member('abc', NameValue) then
  WriteLn('Found with NameValue = ', NameValue)
else
  Writeln('Not Found');
```

Checks to see if 'abc' is a member of the NameDict dictionary. If it is, Member is true and it returns the data we stored with that string.

Declaration

```
procedure Remove(S : String);
```

Purpose

Remove string from dictionary.

Description

If the string S is found in the dictionary, Remove deallocates its space and removes it.

Although the StringDict hash table automatically expands as needed, it does not shrink as strings are removed.

Example

```
NameDict.Add('Nessus', NameValue);
...
NameDict.Remove('Nessus');
```

Adds the string 'Nessus' and a pointer to the NameDict dictionary. Later in the program we remove that string from the dictionary. If 'Nessus' had never been added, Remove would do nothing. A program can call Member first if it is necessary to know whether the string is definitely in the dictionary.

See Also

Clear

Store

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a string dictionary in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes the current size of the hash table followed by a packed image of all the strings and data values for the dictionary. This image may be read by a later call to Load.

To check for errors that may have occurred during the Store, call the stream's GetStatus function as shown in the example below.

The stream registration procedure for a StringDict is StringDictStream.

Example

```
uses OpRoot;
var  NameDict : StringDict;
     NDS : BufIdStream;
     Status : Word;
begin
  if not NameDict.Init then begin
    Writeln('Failed to make StringDict. Status = ', InitStatus); Halt;
  end;
  {Add several names to the StringDic :}
  NameDict.Add('Nessus', 1000);
  NameDict.Add('Beowulf', 1001);
  Status := NameDict.GetStatus;
  if Status <> 0 then begin
    WriteLn('Failed to add Names. Status = ', Status); Halt;
  end;
  {Write the dictionary out to a stream}
  NDS.Init('STRDICT.STM', SCreate, 1024);
  NameDict.Store(NDS);
  Status := NDS.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error, ', Status); Halt;
  end;
  NDS.Done;
  NameDict.Done;
end.
```

Instantiate NameDict, add a couple of strings and some "data" (in this case, just some LongInts), then write the NameDict out to a stream. See Load for an example program to read this stream back in.

An effect similar to calling Store may be achieved with the IdStream.Put method. IdStream.Put is often desirable because it is polymorphic; that is, it automatically determines the appropriate Store method to call based on the stream's data. When a stream is written with Put, the same stream must be read with Get. Get can then determine automatically which Load constructor to call. To use Put, the Store call in the example above may be replaced with

```
NDS.RegisterHier(StringDictStream);
NDS.Put(NameDict);
```

See Also

IdStream.Put
IdStream.RegisterType
Load

Declaration

```
procedure Update(S : String; Value : LongInt);
```

Purpose

Change the data value associated with a string.

Description

Finds the string in the dictionary and changes its value to the newly specified one. If S isn't found, Update does nothing.

Example

```
NameDict.Add('Nessus', 1000);
...
NameDict.Update('Nessus', 2000);
```

Change the data value for 'Nessus' from 1000 to 2000.

See Also

Add

Add / Load

Declaration

```
procedure Add(S : String);
```

Purpose

Add string to set.

Description

Used just like StringDict.Add with the exception that no data value can be specified.

See Also

StringDict.Add

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a string set from a stream.

Description

Used just like StringDict.Load. When using the IdStream.Get methodology, note that StringSet has its own object type (otStringSet) and version code (veStringSet) which should be used in registering the type with the stream. The stream registration procedure for a StringSet is StringSetStream.

See Also

Store StringDict.Load

Declaration
```
function Member(S : String) : Boolean;
```

Purpose
Return true if S is element of set.

Description
Used just like StringDict.Member with the exception that no data value is returned.

See Also
StringDict.Member

Declaration
```
procedure Store(var S : IdStream);
```

Purpose
Store a StringSet in a stream.

Description
Used just like StringDict.Store. When using the IdStream.Put methodology, note that StringSet has its own object type (otStringSet) and version code (veStringSet) which should be used in registering the type with the stream. The stream registration procedure for a StringSet is StringSetStream.

See Also
Load StringDict.Store

AddString / Done

Declaration

```
function AddString(St : String) : Word;
```

Purpose

Add a new string, returning its index, or 0 if error.

Description

Appends the string St to the internal string buffer and assigns it the next available index, which is returned as the function result. The lowest valid index number is 1.

If either the string buffer or the index array is full, AddString will attempt to expand it by allocating additional heap space. If this fails, AddString returns a value of 0.

AddString can add strings ranging from 0 bytes to 255 bytes long. There is no requirement for uniqueness or any kind of check on the contents of the string.

Example

```
var
  S : String;
  I : Word;
...
Write('Enter Msg: ');
ReadLn(S);
if S <> '' then begin
  I := Messages.AddString(S);
  if I = 0 then
    WriteLn('Failed to add message')
  else
    WriteLn('Message added with index ', I);
end;
```

After reading S from the keyboard, we add S to the Messages string array, store its index value and check for success.

See Also

GetString

Declaration

```
destructor Done; virtual;
```

Purpose

Deallocate packed array.

Description

The internal buffer and index arrays are deallocated and the StringArray is destroyed.

Example

```
Messages.Done;
```

Deallocates the heap space used by Messages.

See Also

Init

Declaration

```
function GetString(Which : Word) : String; virtual;
```

Purpose

Return string from packed array.

Description

Which specifies the index number of the string to return. 0 is a special case: it always returns the empty string. If Which is larger than the highest index, GetString also returns the empty string. Otherwise, GetString returns the string originally added with AddString.

GetString is virtual to allow the possibility of a derived type which not only packs the strings together but also compresses them. The WriteText method calls on GetString to write each string to a text file.

Examples

```
WriteLn(Messages.GetString(2));
```

Displays the second string stored in the packed array.

```
WriteLn(Messages.GetString(0));
```

Writes the empty string.

Declaration

```
function GetStringPtr(Which : Word) : StringPtr;
```

Purpose

Return pointer to string from packed array.

Description

Calling this routine will sometimes be more efficient than calling GetString. This routine avoids making an intermediate copy of the string by simply returning a pointer to the copy of the string in the packed array.

If Which is 0 or greater than the total number of strings, GetStringPtr returns the nil pointer.

Do not attempt to modify the string pointed to by GetStringPtr. You are likely to corrupt the packed string buffer by doing so.

Example

```
S := Messages.GetStringPtr(2)^;
```

Assigns the second string stored in the packed array to a string variable.

Init

Declaration

```
constructor Init(StrMax, Amount : Word);
```

Purpose

Allocate space for StrMax strings in Amount space.

Description

The StringArray works by packing strings end-to-end in a byte buffer. In this way, the space wasted by storing varying length strings in a normal array of String is avoided. To provide immediate access to each string, the StringArray also maintains an index, which is just an offset into the packed string buffer for each stored string.

The Init call requires you to provide initial estimates on the size of these two buffers. StrMax specifies the maximum number of strings that can be indexed. Amount specifies the maximum number of bytes (including a length byte) used by all strings.

If you add strings beyond these limits, the StringArray will attempt to expand itself to accommodate them. This may be undesirable, however, because heap fragmentation is guaranteed to occur and because there may not be any heap space left at the time.

Init requires (Amount+2*StrMax) bytes of heap space. If insufficient memory is available, it will fail, returning a status code in the OPROOT global variable InitStatus.

StringArrays may also be allocated by calling the alternate constructors ReadText (to read strings from a text file) and Load. These constructors will often be preferred since the best application of StringArrays is to manage pre-initialized message arrays.

Examples

```
const
  MsgCount = 10; {Total number of messages}
  MsgSize = 20;  {Average size per string}
...
if not Messages.Init(MsgCount, MsgCount*MsgSize) then begin
  WriteLn('Failed to Create StringArray,  Status = ', InitStatus);
  Halt;
end;
```

Creates a packed string array that can hold a maximum of 10 strings with a total of 200 characters. The individual strings can vary in size but the combined total is 200 characters.

```
if not Messages.Init(MsgCount, MsgSize) then begin
  WriteLn('Failed to Create StringArray,  Status = ', InitStatus);
  Halt;
end;
```

Incorrect! While this would certainly work it probably isn't what we want. The second parameter is the total size of the string space, not the size of one message.

See Also

ReadText Load

Declaration
```
constructor Load(var S : IdStream);
```

Purpose
Load a string array from a stream.

Description
S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to StringArray.Store. Load allocates heap space for the index and buffer arrays, then blockreads the previously stored images of those arrays directly into memory.

If Load fails, the constructor will return False and an error code will be stored in the global variable InitStatus.

The stream registration procedure for a StringArray is StringArrayStream.

Example
```
uses OpRoot;
var  Messages : StringArray;
     MS : BufIdStream;
begin
  {Load a StringArray from a stream}
  MS.Init('MSG.STM', SOpen, 1024);
  if not Messages.Load(MS) then begin
    WriteLn('Load error, ', InitStatus); Halt;
  end;
  MS.Done;
  {Display a Message}
  WriteLn(Messages.GetString(2));
  Messages.Done;
end.
```

Creates a StringArray named Messages from a stream created in the Store example. This stream contains three messages. After loading Messages, we display the second string to the screen.

An effect similar to calling Load may be achieved with the IdStream.Get method. See PointerStack.Load for further information.

See Also
Store

Declaration
```
function NumStrings : Word;
```

Purpose
Return number of strings in the packed array.

Description
For a newly created StringArray, this returns 0. Otherwise, it returns the number of strings that have been successfully added.

Example
```
WriteLn('Messages contains ', Messages.NumStrings, ' strings');
```

Displays the number of strings in the string array Messages.

ReadText

Declaration

```
constructor ReadText(FileName : PathStr);
```

Purpose

Initialize a string array from a text file.

Description

Opens the file specified in FileName for reading, then reads the file twice. On the first pass, it counts the number of strings and the amount of storage required. Then it allocates a StringArray of just the right size. It reads the file a second time to initialize the StringArray. Finally, it closes the file.

Because ReadText is dealing with file I/O, there are many possible failure mechanisms. If ReadText fails because of a file error, it returns the value (epFatal + IoResult) in the global variable InitStatus.

Example

```
if not Messages.ReadText('SAMPLE.TXT') then begin
  WriteLn('Failed to Create StringArray,  Status = ', InitStatus);
  Halt;
end;
```

Creates a packed string array filled with text strings from SAMPLE.TXT. SAMPLE.TXT is a normal ASCII text file that might contain the following:

```
Out of memory<CR><LF>
Parameter out of range<CR><LF>
Busy<CR><LF>
```

See Also

Init
Load
WriteText

Declaration

```
function Shrink : Boolean;
```

Purpose

Remove excess array space.

Description

This method shrinks both the index and string buffers to just the size required to hold the strings currently in them. While this method probably reduces the heap usage of a StringArray, it will also usually fragment the heap.

Shrink returns True if successful.

Example

```
Messages.Init(100,2000);
for I := 1 to 3 do begin
  Write('Enter Msg: ');
  ReadLn(S);
  if S <> '' then
    if Messages.AddString(S) = 0 then
      WriteLn('Failed to add message')
    else
      WriteLn('Message added');
end;
if not Messages.Shrink then
  WriteLn('Not enough memory to shrink');
```

The Messages packed string array was initialized far larger than it needed to be. So, after adding all the strings we intend to, we call Messages.Shrink to shrink the size of Messages down to its minimum required size.

Shrink works by creating a new packed array of just the right size and copying the strings from the old array to the new. Ironically, this opens the possibility of running out of memory while trying to shrink the array. Shrink returns False if this happens.

See Also

Init

Declaration

```
function Size : Word;
```

Purpose

Return the number of bytes used by strings.

Description

Returns a measure of how full the string buffer is. For example, if 10 strings each 20 bytes long were added to the StringArray, Size would return 210 (including all the length bytes).

Example

```
WriteLn('Existing messages use ', Messages.Size, ' bytes');
```

Displays the number of bytes currently used by stored strings.

See Also

NumStrings

Store

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Write a string array to a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes the current sizes of the index and string buffers, followed by byte images of the buffers. This sequence may be read by a later call to Load.

To check for errors that may have occurred during the Store, call the stream's GetStatus function as shown in the example below.

The stream registration procedure for a StringArray is StringArrayStream.

Example

```
uses OpRoot;
var  Messages : StringArray;
     MS : BufIdStream;
     Failed : Boolean;
     Status : Word;
begin
  {Create a StringArray (estimate 10 messages at 20 characters each)}
  if not Messages.Init(10, 200) then begin
    WriteLn('Failed to create StringArray. Status = ', InitStatus); Halt;
  end;
  {Add some messages}
  Failed := False;
  if Messages.AddString('Out of Memory') = 0 then
    Failed := True;
  if Messages.AddString('Parameter out of range') = 0 then
    Failed := True;
  if Messages.AddString('Busy') = 0 then
    Failed := True;
  if Failed then begin
    Writeln ('Out of Memory'); Halt;
  end;
  {Write the packed array out to a stream}
  MS.Init('MSG.STM', SCreate, 1024);
  Messages.Store(MS);
  Status := MS.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error, ', Status); Halt;
  end;
  MS.Done;
  Messages.Done;
end.
```

Creates a StringArray named Messages, adds three strings to it, then writes it out to a stream. See Load for an example program to read this stream back in.

An effect similar to calling Store may be achieved with the IdStream.Put method. See PointerStack.Store for more information.

See Also

IdStream.Put
IdStream.RegisterType
Load

Declaration

```
procedure WriteText(FileName : PathStr);
```

Purpose

Write a packed array to a text file.

Description

Scans through the StringArray in index order (starting at index 1) and writes each string to a text file, separated by <CR><LF>.

This method is primarily designed as a utility method, that is, to aid you in writing a program but not to be included in final applications. As such it does not offer the same integrated error checking as most other methods. Check for I/O errors after calling WriteText by reading Turbo Pascal's IoResult function.

Example

```
for I:=1 to 3 do begin
  Write('Enter Msg: ');
  ReadLn(S);
  if S <> '' then
    if Messages.AddString(S) = 0 then
      WriteLn('Failed to add message')
    else
      WriteLn('Message added');
end;
Messages.WriteText('SAMPLE.TXT');
Status := IoResult;
if Status <> 0 then
  WriteLn('Failed to write message file, IoResult = ', Status);
```

Create a packed string array capable of holding three strings. The strings are read from the keyboard. Then write these messages to a text file (presumably, to be read back in with ReadText or by some other program).

OPROOT is set up to compile with I/O checking turned off ($I-). This allows us to do our own I/O checking by referencing IoResult.

See Also

IdStrIdStream.RegisterType ReadText

6. Streams

The term "stream" lends a certain mystique to what is fundamentally a simple concept. The current interpretation of the word apparently originated with the Unix operating system, evolved in the C++ programming language, and was then appropriated with further modification for the Turbo Pascal 5.5 OBJECTS.PAS example unit. Now **Object Professional** adds its own twist to the term. No matter what its etymology, however, a stream provides very useful capabilities in the context of object-oriented programming.

The first thing to know about a stream is that it represents a sequence of bytes, which can be read and written. The interpretation of the bytes is up to the program that reads and writes them. The bytes are often stored in a file, so in this sense a stream can be thought of as a special kind of file. There is really nothing about a stream that limits it to files, however. In fact, **Object Professional** supports streams stored in memory as well as on disk, and this leads to some convenient capabilities for application programmers.

If you like, you can use a stream just like an untyped Turbo Pascal file. You can create a stream, write to it, seek within it, read from it, and close it, just like any file. There's not much point in doing this, though, since normal files work just fine for these purposes.

What makes a stream special is its connection to OOP. A stream makes possible the whole concept of "object-oriented I/O." The typical unit of storage in a stream is one object. That is, the stream holds the data needed to initialize an instance of an object. Of course, it can store more than one instance of the same object type. Actually, the stream can contain instances of *different* object types and a program can still read all these objects from the stream, even without knowing which ones are there or what order they're in.

As used in **Object Professional**'s window system, the stream stores data that describes the shape and position of your windows, what headers they have, all the colors, whether they are pick lists or menus, and so on. Each object has methods that can read and write its own simple components. When an object includes other objects or is derived from another type, it calls upon the methods it inherits from those objects in order to read and write those portions of the stream image.

So why should you care about streams? Perhaps an example from our experience will help to explain. The supplied MAKEMENU and MAKESCRN utility programs read and write libraries of menu and entry screen objects. This makes it easy for you to design an object, save it to disk, and later reread it to make changes. By taking advantage of streams, we didn't need to write one bit of special purpose code to manage these libraries. In fact, a given library can hold both object types and can easily distinguish between the two.

If you write applications that depend on complex menu systems, you'll probably find that loading the menu system definition from a stream will make your programs smaller than building it with source statements. Each Menu.AddItem call generates code, but loading the menu from a stream takes only a single call. Similar arguments apply to packed text screens and all varieties of windows.

Eventually we expect to see Turbo Pascal programmers implementing entire databases filled with objects. You can already find advertisements for leading edge "object-oriented database management" products. What distinguishes such database managers from today's run-of-the-mill programs is that the DBMS doesn't need to know just what each record contains; the methods of each object type are able to classify and decode each record as needed. Streams form the foundation for this futuristic concept.

Implementing streams from the ground up is a bit complicated, but the result is easy enough to use. Before we dive into the details, let's present an example that shows how.

```
program StreamExample;  {EXSTREAM.PAS}
uses
  OpRoot;
const
  otNameData = 1000;          {Name object types}
  otCustData = 1001;
  veNameData = 0;             {Name object versions}
  veCustData = 0;
type
  {An object for manipulating Name and Age}
  NameDataPtr = ^NameData;
  NameData =
    object(Root)
      Name : String[20];
      Age  : Byte;
      constructor Init;
      procedure Display;
      constructor Load(var S : IdStream);
      procedure Store(var S : IdStream);
    end;

  {Descendant from NameData, adds Company name}
  CustData =
    object(NameData)
      CompanyName : String[20];
      constructor Init;
      procedure Display;
      constructor Load(var S : IdStream);
      procedure Store(var S : IdStream);
    end;

var
  Customer : CustData;
  S : BufIdStream;
  I, Status : Word;

  {----- NameData Methods ------------------}
  constructor NameData.Init;
  begin
    WriteLn;
    Write('Name:  ');
    ReadLn(Name);
    Write('Age:   ');
    ReadLn(Age);
  end;

  procedure NameData.Display;
  begin
    WriteLn;
    Writeln(Name,' ',Age);
  end;

  constructor NameData.Load(var S : IdStream);
  begin
    Root.Init;
    Name := S.ReadString;
    S.Read(Age, SizeOf(Age));
```

```
  if S.PeekStatus <> 0 then
    Fail;
end;

procedure NameData.Store(var S : IdStream);
begin
  S.WriteString(Name);
  S.Write(Age, SizeOf(Age));
end;

{$F+}
procedure NameDataStream(SPtr : IdStreamPtr);
begin
  with SPtr^ do begin
    RegisterType(otNameData, veNameData,       {Object code, version number}
                 TypeOf(NameData),             {Object VMT pointer}
                 @NameData.Store,              {Stream write procedure}
                 @NameData.Load);              {Stream read procedure}
  end;
end;
{$F-}

{---- CustData methods -----------------}
constructor CustData.Init;
begin
  NameData.Init;
  Write('Co. Name: ');
  ReadLn(CompanyName);
end;

procedure CustData.Display;
begin
  NameData.Display;
  WriteLn(CompanyName);
end;

constructor CustData.Load(var S : IdStream);
begin
  if not NameData.Load(S) then
    Fail;
  CompanyName := S.ReadString;
  if S.PeekStatus <> 0 then
    Fail;
end;

procedure CustData.Store(var S : IdStream);
begin
  NameData.Store(S);
  S.WriteString(CompanyName);
end;

{$F+}
procedure CustDataStream(SPtr : IdStreamPtr);
begin
  NameDataStream(SPtr);
  with SPtr^ do
    RegisterType(otCustData, veCustData, TypeOf(CustData),
                 @CustData.Store, @CustData.Load);
end;
{$F-}
```

```
begin
  {Create a Stream, check for success}
  if not S.Init('CUSTOMER.STM', SCreate, 1024) then begin
    Writeln('Failed to Create Stream,  Status = ', InitStatus);
    Halt;
  end;

  {Register object hierarchy}
  S.RegisterHier(CustDataStream);
  Status := S.GetStatus;
  if Status <> 0 then begin
    Writeln('Failed to Register objects,  Status = ', Status);
    Halt;
  end;

  {Create and Store three Customer objects, then close the Stream}
  for I := 1 to 3 do begin
    Customer.Init;
    S.Put(Customer);
    Status := S.GetStatus;
    if Status <> 0 then begin
      WriteLn('Failed to Write Customer,   Status = ', Status);
      Halt;
    end;
    Customer.Done;
  end;
  S.Done;

  {Re-open the existing Stream}
  if not S.Init('CUSTOMER.STM', SOpen, 1024) then begin
    Writeln('Failed to Open Stream,  Status = ', InitStatus);
    Halt;
  end;

  {Re-register object hierarchy}
  S.RegisterHier(CustDataStream);
  Status := S.GetStatus;
  if Status <> 0 then begin
    Writeln('Failed to register objects. Status = ', Status);
    Halt;
  end;

  {Retrieve Customers from Stream, Display them, then close the Stream}
  for I := 1 to 3 do begin
    S.Get(Customer);
    Status := S.GetStatus;
    if Status <> 0 then begin
      WriteLn ('Failed to read Customer. Status = ', Status);
      Halt;
    end;
    Customer.Display;
    Customer.Done;
  end;
  S.Done;
end.
```

The example creates a simple hierarchy of two objects: NameData, which stores a person's name and age; and CustData, which adds a CompanyName to the NameData object. Each of these objects includes four methods: normal constructors and (inherited) destructors; plus two methods--Load and Store--especially for dealing with streams.

Load and Store are the key to making object-oriented streams possible. Rather than attempting to build special-purpose code into the stream manager--code that would allow it to read and write specific objects--we offload this task to each object. That way the object itself can initialize its data fields, allocate heap space, and do whatever else it needs to do to instantiate an object.

We've provided Load and Store methods for all the major **Object Professional** types. In this example, we write some new ones to show how easy it can be to use streams with objects of your own.

Both Load and Store take a single var parameter: the stream that's being read or written. The type of the parameter is IdStream. This is the most primitive **Object Professional** stream object, from which various specialized versions are derived. The actual parameter passed may be a buffered file stream (as in this example), a memory stream, a library, or others. To Load and Store, it's all the same because these different kinds of streams are compatible at this level.

The Store method's job is to write the data needed to describe the object. It can access a number of stream methods to help: WriteString writes a Turbo Pascal string to the stream, Write writes the specified number of bytes, and so on. The NameData.Store method writes its Name and Age fields in this way. The CustData.Store method first calls its parent's Store method to write its inherited portion and then writes its own data. This hierarchical technique of storing derived objects is highly recommended. It not only saves work for you, but it subdivides the task along the same boundaries that the objects themselves use.

Load reverses the process by reading from the stream. Load is a constructor, which gives it special capabilities to allocate heap space for dynamic instances and to initialize the virtual method table (VMT) link. Again, CustData.Load first calls its parent's Load method to read the inherited part of the data. If NameData.Load fails, CustData.Load must fail too. Otherwise, CustData.Load reads the rest of the data. Finally, it checks the stream's status variable to detect I/O errors.

Each object with Load and Store methods should also have a "stream registration" procedure associated with it. Here, NameDataStream and CustDataStream are such procedures. These inform the stream of all the object types associated with the given object. You may have wondered how the stream manager can detect what objects are stored in it and decode the sequence of bytes into a meaningful series of objects. Stream registration provides the science behind this apparent magic.

The NameData object doesn't depend on any other objects (except the Root object, which is a special case we'll discuss later). Hence, NameDataStream registers just one type, the NameData object itself. To do so, it calls the stream manager's RegisterType method, which takes several parameters. The first is word-size constant that provides a unique code identifying the object. **Object Professional** reserves the first 1000 codes (0..999) for its own objects. Here, otNameData (which equals 1000) uniquely identifies the NameData object. The stream manager stores this code in the stream when the object is written there; later it can identify the appropriate Load method to call by the value of this code.

The second parameter to RegisterType is the version number of the object. This will allow you to read an old stream file even after you've updated your object definition, just by registering an alternate Load method with a different version number. NameData is a brand new object, so we give it a version number of 0.

The third parameter will always read TypeOf(ObjectBeingRegistered). This calls on a Turbo Pascal system function to return a pointer to the VMT of the object type. This pointer allows

the stream manager to detect what type of object is being written and therefore to call the appropriate Store method (which again helps to explain how streams can store a variety of different object types without having special knowledge of the objects themselves). The stream manager makes an important related assumption, which we'll return to later: all objects to be written in streams *must* be derived from the **Object Professional** Root object.

The final parameters specify the addresses of the Load and Store methods. The stream manager assumes that these routines are *methods* of the described type and with the specified parameter. Any other situation will most likely lead to a program crash. Unfortunately, this is a case where an error cannot be detected at compile time.

Now on to the CustDataStream procedure, which registers the object types for the CustData object. This procedure again takes a hierarchical approach: it calls its parent's registration routine first to register the CustData types, and then registers its own type.

One additional note about the stream registration procedures: These procedures must be compiled under the FAR model (with the {$F+} compiler directive), they must not be nested inside of any other routines, and they must have exactly the one parameter shown. The reason is that these routines are sometimes called indirectly by the stream manager, as will be shown in the example.

The example program declares a global variable S, of type BufIdStream. This derived stream type associates the stream with a DOS file and also provides an in-memory buffer to improve performance while reading and writing. We open the stream file by calling the Init constructor, which takes the filename ('CUSTOMER.STM'), an open mode (SCreate), and a buffer size in bytes (1024).

The next step is to tell the stream manager what data types will be stored in this particular stream. We call the stream's RegisterHier method and pass it the name of our stream registration procedure. The stream manager then calls the specified procedure. You could achieve the same effect with the following statement:

```
CustDataStream(@S);
```

The RegisterHier approach is more consistent with the other object-oriented method calls that surround it.

The RegisterHier call is unlikely to fail. If it does, it's usually the result of a programming error such as assigning the same object code number to two different objects. In addition, the stream manager allocates a small amount of heap space to list the registered objects, so an error could possibly occur due to insufficient memory. In the example we check the stream status just in case.

Three CustData objects are created by calling Customer.Init. Then we call the stream's Put method to write each object to the stream. Put determines the Customer object's type based on the registration information already provided, writes the CustData object and version numbers to the stream, and then transfers control to CustData.Store to finish writing the object. A call to S.GetStatus checks for an I/O error.

After writing the three objects, we close the stream with a call to S.Done. This flushes the in-memory buffer to disk, closes the DOS file, deallocates the buffer, and disposes of its type registration information.

The example proceeds by reopening the stream, this time for reading. It uses the same memory buffer size, although that isn't required. It must re-register the object data types since those

were just disposed. It then calls S.Get to read each customer from the stream. S.Get reads the object and version numbers from the stream and, using the registration information, calls the appropriate Load method. After a successful call to Get, the instance is completely initialized and we can display and destroy it.

It's interesting to note that the stream used in this example could also hold instances of the NameData object type. That type has been indirectly registered via the hierarchical process already described. The stream manager could distinguish between NameData and CustData objects and read or write them correctly to the stream.

The remainder of this section describes the methods associated with streams. The following object types are covered:

```
Root
IdStream
  DosIdStream
    BufIdStream
  MemIdStream
```

We'll save discussion of the Library category of streams for the next section, where we will also provide a complete example program using a Library.

Constants

```
SCreate    = $3C00;          {Create new file}
SOpenRead  = $3D00;          {Read access only}
SOpenWrite = $3D01;          {Write access only}
SOpen      = $3D02;          {Read and write access}
```

Values passed to the DosIdStream and BufIdStream Init methods to indicate how to open a stream file. Each constant specifies the AX register value used with the DOS call that opens the file.

When a stream file is opened on a network, you may need to use other values that take into account the sharing characteristics of the file.

```
otPointerStack          = 001;
  ...
otShadedPField          = 169;
  ...
otReservedForTP         = 999;
```

Unique code numbers for all objects that support stream I/O. You typically won't need to refer to these codes directly since the provided stream registration routines do so. At this time, the last code we've used is 169. Values up to 999 are reserved for TurboPower. You may assign higher codes to your own objects.

```
vePointerStack          = 00;
  ...
veShadedPField          = 00;
```

Version number codes for all objects that support stream I/O. These numbers will be incremented only when an object is modified in such a way as to require a new Load method for it. In that case, the more recent version of the object will get a new version code, and the older version of the object will get a special Load method (Load00, for example) to allow it to be read into the new implementation.

Further details are provided with the IdStream.RegisterOldVersion method.

```
ptNil                   = 0000;  {nil pointer}
ptEmptyOpenCloseProc    = 0001;  {default non-exploding windows}
  ...
ptReservedForTP         = 0999;
ptEntryUserRec          = $FFFF; {used by OPENTRY}
ptFormUserRec           = $FFFF; {used by OPFORM}
```

Pointer codes for use in stream I/O. Just as object types are assigned unique codes in a stream, so are pointers to static code or data. For example, **Object Professional** uses procedure pointers extensively to allow dynamic customization of program behavior. (These are often called "user hooks.") A simple instance of this behavior occurs in the RawWindow object, which associates each window with a procedure that is called when the window is opened. A custom version of the open procedure makes windows "explode" onto the screen. When such a window is stored in a stream and later reloaded, we need a way to associate the custom open procedure with the object. At the same time we need to avoid linking in this custom open procedure when it isn't really used by the program. To solve this problem, the stream manager provides a method you can call to register the address of the custom open procedure. When the stream is written, the registered pointer code is stored in the stream. Later, when the stream is loaded, the pointer code indicates what procedure to use. The process of registering the custom open procedure insures that its code is linked in, even though it isn't directly referred to elsewhere.

Object Professional reserves the codes 0..999 and $FFFF for its own use. You'll probably need to refer to some of these codes when you use streams with objects like StackWindow, PickList, DirList, EntryScreen, and others. The appropriate codes will be mentioned in the documentation for the Load and Store methods of those objects.

Further information about this topic is provided with the IdStream.RegisterPointer method.

Types

```
RootPtr = ^Root;
Root =
  object
    ...
  end;
```

The Root object must be an ancestor of any other object that supports **Object Professional** streams. Root isn't really good for anything by itself--you won't ever want to create an instance of Root. Its declaration guarantees that the VMT link for every derived object occupies the first two bytes of each instance. The stream manager depends on this fact when it writes objects to a stream.

The Root object provides additional practical and theoretical advantages. See Root.Init and Root.Done for details.

```
StreamRegisterProc = procedure (SPtr : IdStreamPtr);
```

Procedure type used by the stream manager when it registers the data types used in a particular stream. A procedure of this type is passed to the IdStream.RegisterHier method. Each "streamable" **Object Professional** object provides one or more procedures of this type. The appropriate procedure is mentioned in the Load and Store methods for each object type.

```
IdStreamPtr = ^IdStream;
IdStream =
  object
    ...
  end;
```

The most primitive declaration of a stream. Because this is an abstract type, you shouldn't declare instances of it. Its properties are inherited by DosIdStream, BufIdStream, MemIdStream, Library, and MemLibrary. Many important methods for type registration are implemented in the IdStream, however, so be sure to read about them in the sections following.

You may note that IdStream is not derived from Root. This circumvents many circular reference problems that would otherwise occur in the IdStream type definition. The only drawback is that you cannot create a stream that stores instances of streams. That's such a mind-boggling concept that we're sure you won't mind, though!

```
DosIdStreamPtr = ^DosIdStream;
DosIdStream =
  object(IdStream)
    ...
  end;
```

Associates a stream with a DOS file. Provides methods to create a new stream file and to read or write an existing one.

```
BufIdStreamPtr = ^BufIdStream;
BufIdStream =
  object(DosIdStream)
    ...
  end;
```

Adds buffering to the DosIdStream. Buffering will provide substantial performance improvements when many small objects comprise the stream.

```
MemIdStreamPtr = ^MemIdStream;
MemIdStream =
  object(IdStream)
    ...
  end;
```

Associates a stream with an area of memory. This is useful when you'd like to link a stream directly into your EXE file. Once a MemIdStream is initialized, it is used exactly like any other stream.

Procedures

OPROOT interfaces the following procedures, each of which is used to register a particular object type with the stream manager. Any of them may be passed as a parameter to the IdStream.RegisterHier method.

```
procedure PointerStackStream(SPtr : IdStreamPtr);
  {-Register all types needed for streams containing PointerStacks}
procedure StaticQueueStream(SPtr : IdStreamPtr);
  {-Register all types needed for streams containing StaticQueues}
procedure SingleListStream(SPtr : IdStreamPtr);
  {-Register all types needed for streams containing SingleLists}
procedure DoubleListStream(SPtr : IdStreamPtr);
  {-Register all types needed for streams containing DoubleLists}
procedure CircularListStream(SPtr : IdStreamPtr);
  {-Register all types needed for streams containing CircularLists}
procedure BitSetStream(SPtr : IdStreamPtr);
  {-Register all types needed for streams containing BitSets}
procedure LargeBitSetStream(SPtr : IdStreamPtr);
  {-Register all types needed for streams containing LargeBitSets}
procedure StringDictStream(SPtr : IdStreamPtr);
  {-Register all types needed for streams containing StringDicts}
procedure StringSetStream(SPtr : IdStreamPtr);
  {-Register all types needed for streams containing StringSets}
procedure StringArrayStream(SPtr : IdStreamPtr);
  {-Register all types needed for streams containing StringArrays}
```

Declaration
```
destructor Done; virtual;
```

Purpose
Destroy a Root object.

Description
Actually, the Root.Done method does nothing at all. It is provided as an abstract hook, guaranteeing that every object derived from Root will have a Done method, even if the method does nothing. As a result, you can feel safe in calling MyInstance.Done whenever you're finished with an **Object Professional** object. Almost all objects derived from Root override this Done method in order to deallocate heap space or otherwise clean up after themselves.

The Done method of every object derived from Root should call Root.Done after doing its own cleanup work. This guarantees that, if **Object Professional** were extended in the future to support virtual memory management or some other unforeseen capability, the Root object would have the opportunity to undo its own actions before the program continued.

Declaration
```
constructor Init;
```

Purpose
Initialize a Root object.

Description
Like Root.Done, Init's function is more theoretical than practical. Init does serve one concrete purpose, however. It resets the global variable InitStatus to 0 so that each individual constructor doesn't need to do so.

The constructor methods of every object derived from Root should call Root.Init before proceeding with their own initialization. Not only does this reset InitStatus, but it also assures that any global initialization required for future extensions of **Object Professional** will occur. It's best if such calls to Root.Init take into account the possibility that the call may fail, by using the following form:

```
if not Root.Init then
  Fail;
```

CodeVerRegistered / Error

Declaration

```
function CodeVerRegistered(Code, Version : Word) : Boolean;
```

Purpose

Return True if specified code and version number are registered.

Description

This method is particularly useful for Library streams, which store the type code and version in each directory entry of the library. By checking whether a particular entry's type is registered, the library can determine in advance whether appropriate initialization was performed.

See Also

Library.GetCodeAndVersion TypeRegistered

Declaration

```
procedure Error(Code : Word); virtual;
```

Purpose

Report an error.

Description

Applications do not call this routine directly. When an IdStream method detects an error, it calls the Error method to report it. The default implementation of this method simply assigns the error code to an internal status variable. An object derived from IdStream may override the Error method to provide different error handling behavior.

See Also

GetStatus PeekStatus

Declaration

```
procedure Get(var R : Root);
```

Purpose

Calls polymorphic Load constructor to read static object R.

Description

Get first reads four bytes from the stream to get the object type code and its version number. It then classifies the object by scanning the stream's registered type list. If a matching type is found, Get transfers control to the Load constructor registered for that type. The Load constructor then reads as many bytes from the stream as it needs to instantiate the object.

Get assumes that the parameter passed to it is a static instance of an object. It calls the Load constructor in such a way as to prevent heap space from being allocated for the instance. The object R can be of any type derived from Root and registered with the stream. Be sure to pass a variable large enough to hold the next object in the stream. Otherwise Get will wipe out memory; the compiler's type checking doesn't prevent you from making a mistake here.

Use GetStatus or PeekStatus to determine whether Get was successful. Errors may occur as the result of I/O problems, because a type was not registered, or because the Load constructor itself failed.

The data read by Get must have been written by Put or PutPtr.

Example

```
...
S.RegisterHier(CustDataStream);
...
S.Get(Customer);
Status := S.GetStatus;
if Status <> 0 then begin
  WriteLn('Failed to Read Customer,  Status = ', Status);
  Halt;
end;
Customer.Display;
```

S.Get calls the Load constructor of Customer to initialize the object. GetStatus retrieves the status code. If that's 0, the object is ready to use.

See Also

GetPtr Put

Declaration

```
function GetPos : Longint; virtual;
```

Purpose

Return the current stream position.

Description

This method is analogous to the system FilePos function for normal files. It returns the current position within the stream. The first position in a stream is 0.

At the IdStream level, this method is abstract (must be overridden). However, the objects that do override it differ only in the details of their implementation.

Example

```
S.Init('TEST.STM', SCreate);
WriteLn(S.GetPos);
```

will write a 0.

See Also

Seek

GetPtr / GetSize

Declaration

```
function GetPtr : RootPtr;
```

Purpose

Calls registered Load constructor to allocate and read object.

Description

This function works like Get, with one major exception: GetPtr calls the associated Load constructor in such a way as to allocate heap space for the object, creating a dynamic instance. This approach is essential when reading from a stream of varying object types, where the order of the types isn't known in advance. GetPtr automatically allocates just the right amount of space for each instance.

GetPtr returns nil if any error occurs. Use the stream's GetStatus or PeekStatus method to determine the reason for the error.

The data read by GetPtr must have been written by Put or PutPtr.

Example

```
S.RegisterHier(CustDataStream);
...
CustomerPtr := S.GetPtr;
if CustomerPtr = nil then begin
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn ('Failed to Read Customer,  Status = ', Status);
    Halt;
  end;
end;
CustomerPtr^.Display;
```

S.GetPtr calls the Load constructor of CustomerPtr and checks the results of that call. This creates a customer object on the heap and places a pointer to that object in CustomerPtr.

See Also

Get PutPtr

Declaration

```
function GetSize : Longint;
```

Purpose

Determine size of stream.

Description

This routine calls two internal virtual methods of the IdStream to return the size of the stream. Given how those methods are implemented for the DosIdStream and BufIdStream objects, the result of GetSize is the same as the file size in bytes. For the MemIdStream, GetSize returns the number of bytes in the memory region allocated to the stream.

Example

```
WriteLn('Stream size: ', S.GetSize);
```

Displays the current file size, in bytes, of the stream S, assuming S is associated with a file.

See Also

MemIdStream.Init

Declaration

```
function GetStatus : Word;
```

Purpose

Return status of stream.

Description

Each IdStream object maintains an internal status variable that stores the current status of the stream. GetStatus provides access to this variable. Like IoResult, GetStatus clears the internal status variable after it returns the current value.

When no error has occurred, GetStatus returns 0. Otherwise it returns an error code as specified in Appendix D.

Examples

```
S.Get(Customer);
Status := S.GetStatus;
if Status <> 0 then begin
  WriteLn ('Failed to Read Customer,  Status = ', Status);
  Halt;
end;
```

Checks the status of the last operation(s). Referencing the function GetStatus also clears the status.

```
if S.GetStatus <> 0 then
  WriteLn('Failed to Read Customer, Status = ', S.GetStatus);
```

Incorrect! Reading status clears it. If you need to reference the status more than once, move it into a variable or call PeekStatus instead.

See Also

Error

Declaration

```
function PeekStatus : Word;
```

Purpose

Return status and leave internal variable unchanged.

Description

Works just like GetStatus except that it doesn't clear the internal status variable.

Example

```
constructor NameData.Load(var S : IdStream);
begin
  if not Root.Init then
    Fail;
  Name := S.ReadString;
  S.Read(Age, SizeOf(Age));
  if S.PeekStatus <> 0 then
    Fail;
end;
```

Peeks at the status of the last stream operation without clearing the status value. This is especially useful in constructors where you want to Fail the constructor, but you want to leave the status value available to the caller.

See Also

Error

PointerRegistered / Put

Declaration

```
function PointerRegistered(Ptr : Pointer) : Boolean;
```

Purpose

Return True if specified pointer is registered with stream.

Description

See the RegisterPointer method and the Constant declarations for this section for a discussion of the concepts of pointer registration. PointerRegistered is used to determine whether a specific pointer value has been registered with a stream.

Example

```
if not S.PointerRegistered(@VenetianOpenProc) then
  S.RegisterPointer(1000, @VenetianOpenProc);
```

Check if a pointer to the VenetianOpenProc routine is already registered; if not, register it now. Note that referring to @VenetianOpenProc guarantees that the compiler's smart linker will pull in the code for the specified routine. Be careful about using PointerRegistered to refer to routines that are needed only optionally.

See Also

RegisterPointer

Declaration

```
procedure Put(var R : Root);
```

Purpose

Polymorphically call Store method to write object R.

Description

Put examines the first two bytes of the R instance variable. For objects derived from Root, these bytes contain the DS-relative offset of the object's virtual method table. Put classifies the object by scanning the stream's registered type list. Assuming the type is found, Put writes the object's type and version numbers to the stream (4 bytes total), then transfers control to the Store method registered for that type. The Store method writes as many bytes to the stream as it needs to describe the object fully.

The parameter passed to Put may be either a static or dynamic instance of an object. The object type must have Root as a direct or indirect ancestor. Put and PutPtr are essentially the same routine, differing only in the syntax by which they are called.

Use GetStatus or PeekStatus to determine whether Put was successful. Errors may occur as the result of I/O problems, because a type was not registered, or because the Store method itself failed.

Data written by Put may be read from the stream using Get or GetPtr.

Example

```
S.RegisterHier(CustDataStream);
...
S.Put(Customer);
Status := S.GetStatus;
if Status <> 0 then begin
  WriteLn('Failed to write Customer.  Status = ', Status); Halt;
end;
```

S.Put calls the Store method of Customer (of type CustData).

See Also

Get PutPtr

Declaration

```
procedure PutPtr(RPtr : RootPtr);
```

Purpose

Calls polymorphic Store method to write object at RPtr^.

Description

Works just like Put. The only difference is the syntax by which it is called. For a static instance, the following two statements are equivalent:

```
S.Put(Customer);
S.PutPtr(@Customer);
```

For a dynamic instance, the following have identical effects:

```
S.Put(CustomerPtr^);
S.PutPtr(CustomerPtr);
```

See Also

GetPtr Put

ReadPointer

Declaration

```
function ReadPointer : Pointer;
```

Purpose

Reads pointer from stream.

Description

ReadPointer reads the next 4 bytes from the stream. The first 2 bytes are the registered code of the pointer, and the second 2 are a special version code ($FFFF) indicating that the first two bytes refer to a pointer and not to an object type. ReadPointer scans the registered type list to find a physical pointer corresponding to these codes. If it finds one, it returns the physical pointer. Otherwise it returns nil.

See the RegisterPointer method and the Constant declarations for this section for a discussion of the concepts of pointer registration.

Example

```
type
  NameData =
    object(Root)
      Name : String[20];
      BufPtr : Pointer;
      constructor Init;
      procedure Display;
      constructor Load(var S : IdStream);
      procedure Store(var S : IdStream);
    end;
...
constructor NameData.Load(var S : IdStream);
begin
  if not Root.Init then
    Fail;
  Name := S.ReadString;
  BufPtr := S.ReadPointer.
  if S.PeekStatus <> 0 then
    Fail;
  if BufPtr = nil then
    Fail;
end;

var
  Name : NameData;
...
S.RegisterHier(NameDataStream);
GetMem(TempPtr, 100);
S.RegisterPointer(1000, TempPtr);
...
Name.Load(S);
```

The NameData object includes a pointer, BufPtr, that points to a data region that is normally allocated outside of its own constructor. Whoever created the stream of NameData objects will have registered the address of a region that its BufPtr referred to. The example allocates a region of the correct size (TempPtr^) and registers the pointer with a pointer code of 1000. The Load constructor's call to ReadPointer will associate the value of TempPtr with the object's data field BufPtr.

See Also

ReadUserPointer
RegisterPointer
WritePointer

Declaration

```
procedure ReadRange(var BufStart, BufBeyond);
```

Purpose

Read Ofs(BufBeyond)-Ofs(BufStart) bytes into BufStart.

Description

This method is useful for initializing a specified range of data fields of an object within a Load constructor. Pass the first field to read as the BufStart parameter, and the field just past the last one as BufBeyond. The number of bytes read is Ofs(BufBeyond)-Ofs(BufStart).

The parameters passed to ReadRange should always be in the same segment (that is, both local variables, both global, or both part of the same heap block. This condition is met automatically if the two parameters are data fields within the same object instance.

The compiler cannot perform any kind of type checking on the parameters passed to ReadRange. Be cautious!

Example

```
NameData =
  object(Root)
    Name : String[20];
    Age : Byte;
    Other : array[1..10] of Word;
    Junk : Byte;
    constructor Init;
    procedure Display;
    constructor Load(var S : IdStream);
    procedure Store(var S : IdStream);
  end;
...
constructor NameData.Load(var S : IdStream);
begin
  if not Root.Init then
    Fail;
  Name := S.ReadString;
  S.ReadRange(Age, Junk);
  if S.PeekStatus <> 0 then
    Fail;
end;
```

Shows an object definition and a constructor to instantiate such an object from a stream. ReadRange reads data from the stream directly into the object, starting at the Age field and continuing up to, but not including, the Junk field. To initialize the Junk field would require another call, probably to the Read method.

See Also

DosIdStream.Read WriteRange

ReadString

Declaration

```
function ReadString : String;
```

Purpose

Read next variable length string from stream.

Description

Reads just enough bytes to initialize the actual length of a string. It does so by first reading the length byte, then by reading the number of bytes specified in the length byte. Such a string must have been written to the stream by the WriteString method. The size of a given stream can be minimized by using these methods instead of just writing the complete size of each string, including the unused portion.

This method correctly reads empty (zero length) strings.

Example

```
type
  NameData =
    object(Root)
      Name : String[20];
      constructor Init;
      procedure Display;
      constructor Load(var S : IdStream);
      procedure Store(var S : IdStream);
    end;
...
constructor NameData.Load(var S : IdStream);
begin
  if not Root.Init then
    Fail;
  Name := S.ReadString;
  if S.PeekStatus <> 0 then
    Fail;
end;
```

Shows an object definition and a constructor to instantiate it from a stream. ReadString reads the length byte, then the actual number of characters for this instance of Name.

See Also

WriteString

Declaration

```
function ReadUserPointer(DefaultPtr : Pointer) : Pointer;
```

Purpose

Reads pointer and assigns DefaultPtr if none registered.

Description

Works like ReadPointer, with one exception. Whereas ReadPointer returns nil if it determines that no pointer was registered to match the code contained in the stream, ReadUserPointer instead returns DefaultPtr. This method is useful within the Load constructor of objects that have optional user hooks. For example, the Load method for the CommandWindow calls ReadUserPointer to read the address of the window's error handling procedure. If you have registered a custom error handler, ReadUserPointer will return the address of your custom routine. If not, ReadUserPointer returns the address of the default handler, DefaultErrorProc, which we pass as the parameter to ReadUserPointer.

ReadUserPointer is really just a convenience routine which saves a couple of lines of code.

See the RegisterPointer method and the Constant declarations for this section for a discussion of the concepts of pointer registration.

Example

```
@CustomProc := S.ReadUserPointer(@DefaultProc);
```

Reads a pointer code from the stream. If that pointer code is registered, the pointer value is moved into the CustomProc variable, which is a procedure or function type. Otherwise, the address of DefaultProc is used.

See Also

RegisterPointer

WriteUserPointer

RegisterHier

Declaration

```
procedure RegisterHier(RProc : StreamRegisterProc);
```

Purpose

Call a supplied routine to register a hierarchy of object types.

Description

Object-oriented stream I/O depends heavily on the user of a stream to register the various object types and pointers that will be part of the stream. To make registration as easy as possible, the author of any "streamable" object should write a procedure that registers all the types required by that object, including itself. Such procedures are provided for all streamable objects in **Object Professional**.

RegisterHier calls one of those procedures on behalf of a particular stream. The procedure it calls must take a particular form:

```
{$F+}
procedure XxxStream(SPtr : IdStreamPtr);
```

To call RegisterHier with this procedure would require the following statement:

```
S.RegisterHier(XxxStream);
```

Object Professional follows a consistent naming convention for its stream registration procedures. Each one is just the name of the object immediately followed by the word "Stream". Hence, PointerStackStream, RawWindowStream, EntryScreenStream, MenuStream, and so on.

If a stream contains more than one object type, more than one RegisterHier call may be required. If the objects in the stream use procedure pointers or other user-defined pointer types, calls to RegisterPointer or RegisterUserPointer may also be needed.

RegisterHier will typically be called immediately after the stream is initialized via a successful call to Init.

RegisterHier will generate an error if the registration procedure attempts to assign the same type code to more than one object or pointer, or if there is insufficient heap space to accommodate a new link in the registered ID list.

Example

```
{Create a Stream, check for success}
if not S.Init('CUSTOMER.STM', SCreate, 1024) then begin
  Writeln('Failed to Create Stream,  Status = ', InitStatus);
  Halt;
end;

{Register object hierarchy}
S.RegisterHier(CustDataStream);
Status := S.GetStatus;
if Status <> 0 then begin
  Writeln ('Failed to Register objects,  Status = ', Status);
  Halt;
end;
```

After creating a new stream file, calls RegisterHier to register the types associated with the CustData object. CustDataStream is the same procedure defined in this section's introductory program.

See Also

RegisRegisterPointer RegisterType

Declaration

```
procedure RegisterOldVersion(Code, Version : Word;
                             TypePtr, LoadPtr : Pointer);
```

Purpose

Registers an alternate loader for an old version of an object type.

Description

This procedure, in combination with the overall implementation of the IdStream, solves a potential problem with object-oriented I/O. Suppose we write a stream today that contains an instance of object A. Tomorrow we modify object A to include a new data field which must be written to the stream. Can we read the old stream? Probably not. If we modified the old Load routine to somehow detect the difference between the two formats, then we could read it, but eventually we'd have a large messy Load routine that implemented the entire history of the object.

The IdStream lets us define more than one Load constructor for a given object. Each instance written to a stream with Put stores not only an object type code but also a version number code. When the stream is read with Get, the stream manager first looks at the type code and then at the version number to decide which Load routine is appropriate. In this way, each Load routine can be optimized to handle a specific format. If certain version numbers are considered obsolete they need not be registered with the stream manager, saving code space. Note that old Load routines should provide appropriate values for data fields added since the stream was written.

Also note that RegisterOldVersion doesn't take the address of an alternate Store routine as a parameter. Instances written to a stream must always be written in the current format, using a single Store method per object.

The Code number here must be the same as that passed to RegisterType for the same object. By convention, the oldest Version number is 0 and successive new ones increment by one. The value passed for TypePtr is always TypeOf(ObjectBeingRegistered). LoadPtr is the address of the Load constructor for the old version.

RegisterOldVersion will generate an error if it specifies a conflicting code number or if insufficient memory is available to add a link to the registration list.

Example

```
{$F+}
procedure NameDataStream(SPtr : IdStreamPtr);
begin
  with SPtr^ do begin
    RegisterType(otNameData, 1,              {Object code, version number}
                 TypeOf(NameData),           {Object size}
                 @NameData.Store,            {Stream write procedure}
                 @NameData.Load);            {Stream read procedure}
    RegisterOldVersion(otNameData, 0,        {Object code, version number}
                       TypeOf(NameData),     {Object size}
                       @NameData.Load0);     {Stream read procedure}
  end;
end;
{$F-}
```

This stream registration procedure assumes that we will sometimes have to read an older version of the NameData object. Hence, it registers an object of the same type but with a different version number and supplies the Load0 constructor to read this object. If any of the older object types are encountered in a stream, the proper Load procedure will be called to read them. NameData.Load0 must be declared just like any other Load method, that is:

```
constructor Load0(var S : IdStream);
```

See Also

RegisterType TypeVerRegistered

RegisterPointer

Declaration

```
procedure RegisterPointer(Code : Word; Ptr : Pointer);
```

Purpose

Registers one pointer for IdStream I/O.

Description

Object Professional object types make extensive use of pointers. In many cases, the use of pointers has no impact on stream I/O. For example, when heap space is allocated for a window buffer, the object's instance stores a pointer to the buffer. The stream doesn't need to store the pointer, though, since a new buffer will be allocated when the object is reloaded from the stream. In general, when an object's method allocates heap space using New or GetMem, there is no need to store that pointer in the stream, and therefore no need to register the pointer with the stream manager.

The connection between pointers and stream I/O becomes important when an object stores a pointer to something that it doesn't allocate for itself. Prime examples include pointers to procedures and to externally defined data structures. In this case, when an object is reloaded we must assure that the pointer ends up pointing to the same logical location it did when the object was stored. Storing a physical memory address in the stream wouldn't be the correct approach; if you installed a new TSR or even made a slight program modification before rereading the stream, the absolute address would no longer be valid.

Hence the need for pointer registration. Instead of storing a physical address in the stream we store a code number. The code number is associated with a physical address through the process of registration. Because the writer and the reader of the stream perform independent registration, differences in program load address and code alignment are automatically accounted for.

A subtler advantage of pointer registration becomes apparent when one considers the effects on Turbo Pascal's smart linker. Suppose for example that a particular object uses a procedure pointer to allow three different forms of behavior depending on which of three procedures the pointer selects. (The PickList object does exactly this with its vertical, horizontal, and snaking list orientations.)

A workable approach to storing such an object in a stream would be for the Load constructor to use a case statement. A code number stored in the stream would select among the labels of the case statement to store a pointer to the appropriate routine. Unfortunately, this approach forces the smart linker to pull in the code of all three procedures even though only one may be needed.

With pointer registration, the linker will pull in only those routines explicitly named during registration. Even more importantly, pointer registration allows the use of pointers to routines that weren't originally supplied with the object. Extending the case statement approach would require editing the object's source code to add a new label.

With that as introduction, the use of the RegisterPointer routine is straightforward. Code is the value actually written to the stream--it symbolizes the pointer passed in the Ptr parameter.

TurboPower reserves the codes 0..999 and $FFFF for its own use. You'll probably be using these codes more than any others because we have assigned them to named constants for the built-in procedure pointers in **Object Professional**. Each object that needs them will describe the appropriate constants.

Note that the stream ID codes used by pointers are independent of those used for object types. You could have a pointer code of 1000 and an object type code of 1000 in the same stream, for example.

As with RegisterType, RegisterPointer will generate an error if the specified code is in conflict with an already-registered pointer, or if insufficient heap space is free to expand the registration list.

Example

```
type
  WriteProc = procedure (S : String);
  StringWriter =
    object(Root)
      WrProc : WriteProc;
      St : String;
      constructor Init(S : String; W : WriteProc);
      procedure WriteIt;
      procedure Store(var S : IdStream);
      constructor Load(var S : IdStream);
    end;

{$F+}
procedure TextWrite(S : String);
begin
  Write(S);
end;
{$F-}

constructor StringWriter.Init(S : String; W : WriteProc);
begin
  if not Root.Init then
    Fail;
  WrProc := W;
  St := S;
end;

procedure StringWriter.WriteIt;
begin
  WrProc(St);
end;

procedure StringWriter.Store(var S : IdStream);
begin
  S.WriteString(St);
  S.WritePointer(@WrProc);
end;

constructor StringWriter.Load(var S : IdStream);
begin
  if not Root.Init then
    Fail;
  St := S.ReadString;
  @WrProc := S.ReadPointer;
  if S.PeekStatus <> 0 then
    Fail;
  if @WrProc = nil then begin
    WriteLn('No write procedure was registered');
    Fail;
  end;
end;

var
  S : DosIdStream;
  W : StringWriter;
```

```
begin
  {Create a StringWriter}
  W.Init('hello world', TextWrite);

  {Write the stream}
  S.Init('TEST.STM', SCreate);
  S.RegisterType(1000, 0, TypeOf(StringWriter),
                 @StringWriter.Store, @StringWriter.Load);
  S.RegisterPointer(1000, @TextWrite);
  S.Put(W);
  S.Done;

  {Reread the stream}
  S.Init('TEST.STM', SOpen);
  S.RegisterType(1000, 0, TypeOf(StringWriter),
                 @StringWriter.Store, @StringWriter.Load);
  S.RegisterPointer(1000, @TextWrite);
  S.Get(W);
  S.Done;

  W.WriteIt;
end.
```

Defines an object that depends upon a procedure variable to write to the screen. Alternate procedures might write in graphics mode, sideways, etc. The address of the desired write procedure is registered with code 1000 prior to writing and reading the stream. Note that the Load constructor checks whether ReadPointer returned a value of nil. This occurs when no pointer was registered. The example ignores other potential errors, however.

See Also

PointerRegistered RegisterType

Declaration

```
procedure RegisterType(Code, Version : Word;
                       TypePtr, StorePtr, LoadPtr : Pointer);
```

Purpose

Registers one object type of specified version for IdStream I/O.

Description

The introduction to this section describes the rationale for type registration. To recap briefly: by knowing a code number that represents each object type, the stream manager is able to direct the writing and reading of each instance to the Store and Load methods of the appropriate object type. In this way, the stream manager doesn't need to know the details of any of the objects it manages.

The RegisterType method makes the connection between a Code and Version number, the type of an object, and the Store and Load methods for that object. The stream manager assumes that any registered object type descends (directly or indirectly) from the Root object. RegisterType cannot enforce this requirement; that is your responsibility.

The Code number must be unique to a given object for a given stream. To avoid conflicts, TurboPower reserves the codes 0..999 and assigns a code to all the objects in **Object Professional**. Named constants for each object are found in OPROOT.PAS and are described in the manual with the stream registration routines for each object. Each constant starts with the letters 'ot' and ends with the name of the object type.

Each object type may have one or more versions (see RegisterOldType for more discussion). The Version number passed to RegisterType should be the most recent (which by convention means the largest). TurboPower reserves the version number $FFFF for internal use. Any other Word is acceptable. TurboPower defines constants for the current version of each object. Each such constant starts with the letters 've' and ends with the name of the object type.

TypePtr must be a pointer to the virtual method table (VMT) of the object. Every object that can be stored in a stream will have a VMT, because it descends from the Root object which has one. The Turbo Pascal function TypeOf(ObjectType) is used to obtain a pointer to the VMT of object ObjectType.

StorePtr is the address of the Store method that the stream manager will call. Store must be a method of the object being registered (or perhaps of its ancestor) and it must have the following declaration:

```
procedure Store(var S : IdStream);
```

The stream manager cannot enforce this requirement. Use the following syntax to obtain the address of ObjectType's Store method:

```
@ObjectType.Store
```

LoadPtr is the address of the Load constructor called by the stream manager. The Load method must have the following declaration:

```
constructor Load(var S : IdStream);
```

RegisterType will generate an error if the specified code is in conflict with an already-registered object, or if insufficient heap space is free to expand the registration list. Use the GetStatus method to check for errors.

Example

See the example at the beginning of this section and also for RegisterPointer.

See Also

RegisterOldType
RegisterPointer
TypeIsRegistered

Seek / TypeRegistered

Declaration

```
procedure Seek(Pos : Longint);
```

Purpose

Set the current stream position to Pos, relative to the beginning of stream.

Description

Works much like the Seek procedure defined for normal files. After calling Seek, the next bytes read or written will come from or go to the specified Pos. The first byte in a stream is at position 0.

When a stream is used for object storage, this routine typically isn't useful because the size of an object's image may vary. However, the Library objects make good use of Seek to allow random access to individual objects in a stream.

Example

```
S.Seek(100);
```

Moves the file pointer to the 100th byte in the stream.

See Also

GetPos

Declaration

```
function TypeRegistered(TypePtr : Pointer) : Boolean;
```

Purpose

Return True if specified type is registered for store and load.

Description

TypePtr is a pointer to the virtual method table (VMT) of an object. The VMT pointer is obtained by using the Turbo Pascal TypeOf function.

Be aware that referring to an object type's VMT causes the compiler's smart linker to link in the code for all virtual methods of the object, whether or not any of those methods are actually called.

Example

```
if not TypeRegistered(TypeOf(NameData)) then
  RegisterType(otNameData, veNameData,       {Object code, version number}
               TypeOf(NameData),             {Object size}
               @NameData.Store,              {Stream write procedure}
               @NameData.Load);              {Stream read procedure}
```

If NameData isn't yet registered, this will register it. Actually, the check isn't necessary since RegisterType quietly ignores duplicate requests to register objects. Actual applications of TypeRegistered will occur in more complicated situations.

See Also

RegisterType

Declaration

```
function TypeVerRegistered(TypePtr : Pointer; Version : Word) : Boolean;
```

Purpose

Return True if specified type and version is registered for load.

Description

TypePtr is a pointer to the virtual method table (VMT) of an object. The VMT pointer is obtained by using the Turbo Pascal TypeOf function. Version is a version code to check.

Example

```
if not TypeVerRegistered(TypeOf(NameData), vePrevVer) then
  RegisterOldVersion(otNameData, vePrevVer,  {Object code, version number}
                     TypeOf(NameData),       {Object size}
                     @NameData.Load);        {Stream read procedure}
```

If specified version of NameData isn't registered yet, this will register it.

See Also

TypeRegistered

Declaration

```
procedure WritePointer(Ptr : Pointer);
```

Purpose

Writes identifier for pointer to stream.

Description

Writes four bytes to the stream. The four bytes consist of the registered pointer code (2 bytes) followed by $FFFF, which is a version number reserved for pointers.

WritePointer will generate an error if the specified pointer is not registered. It will also generate errors for I/O problems. Check for errors by calling GetStatus.

Further information about pointer registration is provided with the RegisterPointer routine.

Example

The example for RegisterPointer shows the Store method using WritePointer to write a procedure variable to a stream.

See Also

RegisterPointer

WriteRange / WriteString

Declaration

```
procedure WriteRange(var BufStart, BufBeyond);
```

Purpose

Write Ofs(BufBeyond)-Ofs(BufStart) bytes to stream.

Description

This method is useful for storing a specified range of data fields of an object within a Store method. Pass the first field to write as the BufStart parameter, and the field just past the last one as BufBeyond. The number of bytes written is Ofs(BufBeyond)-Ofs(BufStart).

The parameters passed to WriteRange should always be in the same segment (that is, both local variables, both global, or both part of the same heap block). This condition is met automatically if the two parameters are data fields within the same object instance.

The compiler cannot perform any kind of type checking on the parameters passed to WriteRange. Be cautious!

Example

```
NameData =
  object(Root)
    Name : String[20];
    Age : Byte;
    Other : array[1..10] of Word;
    Junk : Byte;
    constructor Init;
    procedure Display;
    constructor Load(var S : IdStream);
    procedure Store(var S : IdStream);
  end;
...
procedure NameData.Store(var S : IdStream);
begin
  S.WriteString(Name);
  S.WriteRange(Age, Junk);
end;
```

S.WriteRange writes data from the object to the stream starting with the first byte of Age and continuing up to, but not including, the first byte of Junk.

See Also

ReadRange

Declaration

```
procedure WriteString(S : String);
```

Purpose

Write a variable length string to stream.

Description

Writes just enough bytes to store the actual length of a string. The actual number of bytes written is Length(S) + 1. The size of a given stream is minimized by using WriteString instead of writing the complete size of each string, including the unused portions.

Example

```
S.WriteString(Name);
S.WriteString('testing');
```

Writes the actual number of bytes in the two strings, including their length bytes.

See Also

ReadString

Declaration

```
procedure WriteUserPointer(Ptr : Pointer; DefaultCode : Word);
```

Purpose

Writes Ptr code if registered, else DefaultCode.

Description

Works like WritePointer, with one exception. If the pointer Ptr is not registered, WriteUserPointer writes the DefaultCode to the stream instead of generating an error. This behavior is useful for objects that support an optional "user hook" procedure, for example a pre-edit routine in an EntryScreen. In that case, if there is no user procedure, the code for a default do-nothing routine is written instead.

Further information about pointer registration is provided with the RegisterPointer routine.

Example

```
procedure StringWriter.Store(var S : IdStream);
begin
  S.WriteString(St);
  S.WriteUserPointer(@WrProc, ptDefaultWrite);
end;
```

Writes the code (ptDefaultWrite) for a default write routine if the actual WrProc has not been registered.

See Also

RegisterPointer WritePointer

Done / Init

Declaration

```
destructor Done; virtual;
```

Purpose

Close file and dispose stream.

Description

Uses a DOS service to close the file previously associated with the stream. Then calls IdStream.Done to dispose of the registration list.

Example

```
S.Init('TEST.STM', SCreate);
...
S.Done;
```

Closes the file TEST.STM and disposes of the stream.

Declaration

```
constructor Init(FileName : String; Mode : Word);
```

Purpose

Open DOS file in specified mode and initialize stream.

Description

FileName is any valid DOS pathname. The file can be opened in any valid DOS mode. Mode is a value that will be stored in the AX register prior to calling DOS. Predefined constants available for Mode include:

```
SCreate    = $3C00;          {Create new file}
SOpenRead  = $3D00;          {Read access only}
SOpenWrite = $3D01;          {Write access only}
SOpen      = $3D02;          {Read and write access}
```

When a stream file is created, it is given the "normal" attribute (i.e., it is not marked read-only or hidden).

The "Open" modes require that the specified file already exist. Additional open modes can be specified to allow sharing of the stream file on a network. See a DOS reference manual for details.

If the constructor fails, it will return False and an error code will be available in the global variable InitStatus.

Example

```
if not S.Init('NAME.STM', SCreate) then begin
  Writeln ('Failed to Create Stream,  Status = ', InitStatus);
  Halt;
end;
```

Initializes a DosIdStream and creates the file NAME.STM.

See Also

IdStream.Init

Declaration

```
procedure Read(var Buf; Count : Word); virtual;
```

Purpose

Read bytes from DOS file.

Description

Works much like the BlockRead function for untyped file variables. Count bytes of data are read from the stream into memory starting at the address specified by Buf. Because Buf is an untyped parameter, it is your responsibility to assure that the region it refers to is large enough to hold Count bytes. When possible, use the Turbo Pascal SizeOf operator to determine the byte count. If Count is 0, Read does nothing.

Other methods in the IdStream object (ReadRange, ReadString, ReadPointer) call upon this virtual method to read raw bytes.

If an error is pending when Read is called, it does nothing. This means that it is often safe to do a series of reads and to call GetStatus only at the end of the sequence.

If an error occurs during the Read call, it fills Buf with Count bytes of zeros and then generates an error.

Error codes generated by Read are consistent with DOS (and Turbo Pascal) I/O codes, with the addition of an error prefix code. See Appendix D for details.

Examples

```
S.Read(W, SizeOf(Word)); {W is a word variable}
```

Initializes a word variable, W.

```
S.Read(St, SizeOf(St));
```

Initializes a string variable, St.

See Also

ReadRange
ReadString
Write

Declaration

```
procedure Truncate; virtual;
```

Purpose

Truncate file at current position.

Description

Works just like the standard Truncate procedure applied to a normal file. Uses DOS function $40 with CX = 0 to truncate the file at the current position.

Does nothing if there is a pending stream error.

Example

```
S.Seek(100);
S.Truncate;
```

Truncates the stream to contain 100 bytes.

Write

Declaration

```
procedure Write(var Buf; Count : Word); virtual;
```

Purpose

Write bytes to DOS file.

Description

Works much like the BlockWrite function for untyped file variables. Count bytes of data are written to the stream from memory starting at the address specified by Buf. Because Buf is an untyped parameter, it is your responsibility to assure that the region to which it refers holds Count initialized bytes. When possible, use the Turbo Pascal SizeOf operator to determine the byte count. If Count is 0, Write does nothing.

Other methods in the IdStream object (WriteRange, WriteString, WritePointer) call upon this virtual method to write raw bytes.

If an error is pending when Write is called, it does nothing. This means that it is often safe to do a series of writes and to call GetStatus only at the end of the sequence.

Error codes generated by Write are consistent with DOS (and Turbo Pascal) I/O codes, with the addition of an error prefix code. See Appendix D for details.

Example

```
S.Write(W, SizeOf(W));
```

Write a word variable.

```
S.Write(St, SizeOf(St));
```

Writes a string variable to a stream. Note that this writes the complete size of the string rather than just the used portion. Use WriteString to write just the portion currently in use.

See Also

Read
WriteRange
WriteString

Declaration
```
destructor Done; virtual;
```

Purpose
Flush and close file, deallocate buffer, dispose stream.

Description
If the stream file was opened for writing, and bytes remain in the stream's memory buffer, these are written to disk. The DOS file is then closed. The stream's memory buffer is deallocated, and the IdStream itself is disposed.

Example
```
S.Init('TEST.STM', SOpen, 1024);
...
S.Done;
```
Flushes and closes a stream file that was opened for read and write with a 1024 byte buffer.

See Also
Init

Declaration
```
procedure Flush; virtual;
```

Purpose
Flush buffer.

Description
If there are no bytes pending in the memory buffer, the buffer pointers are reset and Flush exits.

If bytes remaining in the memory buffer were placed there by reading from the stream, the buffer pointers are reset (to indicate an empty buffer) and the physical file pointer is positioned just past the last buffered byte that was actually used.

If bytes remaining in the memory buffer were placed there by writing to the stream, those bytes are written to disk and the buffer pointers are reset to indicate an empty buffer.

Flush does nothing if an error is pending.

In most cases, flushing is done automatically (e.g., when the write buffer fills or the stream is closed). The Flush method is made available for transaction-oriented stream activities or other advanced applications.

Example
```
S.Flush;
```
Flushes the I/O buffer to or from disk, whichever is required.

Init / Read / Write

Declaration

```
constructor Init(FileName : String; Mode, Size : Word);
```

Purpose

Open file with specified Mode and allocate buffer of Size bytes.

Description

Works like DosIdStream.Init with the addition of a memory buffer used to enhance the performance of reads and writes. Size bytes of heap space are allocated for the buffer.

When writing to a BufIdStream, the bytes are normally transferred directly into the buffer. Only when the buffer fills is a physical disk write performed.

Similarly, when reading from a BufIdStream, a full buffer of information is read at a time, and portions of the buffer are returned to fill specific requests.

Individual reads and writes may request more bytes than the size of the memory buffer.

A buffer size of 1024 to 4096 bytes is usually a good choice for optimizing performance without spending too much memory.

If the constructor fails, it returns False and stores an error code in the global variable InitStatus.

Example

```
if not S.Init('NAME.STM', SCreate, 1024) then begin
  Writeln('Failed to create Stream. Status = ', InitStatus); Halt;
end;
```

Initializes a BufIdStream and creates the file NAME.STM. This also allocates a buffer of 1024 bytes.

See Also

Done

Declaration

```
procedure Read(var Buf; Count : Word); virtual;
```

Purpose

Read, using buffer.

Description

Works just like DosIdStream.Read, with the exception that requests are filled from the memory buffer whenever possible. When a physical disk read is required, the buffer is filled if possible.

See Also

DosIdStream.Read

Declaration

```
procedure Write(var Buf; Count : Word); virtual;
```

Purpose

Write, using buffer.

Description

Works just like DosIdStream.Write, with the exception that writes are temporarily stored in the memory buffer until it fills or is flushed. The Count parameter may exceed the size of the buffer with no ill effects.

See Also

DosIdStream.Write

Declaration

```
constructor Init(BasePtr : Pointer; Size : Word);
```

Purpose

Initialize stream to point to a location in memory.

Description

This form of stream is not associated with a DOS file. Instead, it reads or writes its sequence of bytes to a region in memory specified by this constructor call.

The most common use of a MemIdStream will be to link a file-based stream into a program, in order to avoid distributing a separate file for the stream. This is done by using Borland's BINOBJ utility, as follows. Suppose you have a stream file called STUFF.STM and that the file size (reported by DIR) of this file is 2000 bytes. First you must convert the stream file into an OBJ file by running BINOBJ at the DOS command line:

```
BINOBJ STUFF.STM STUFF.OBJ STUFF
```

This command line says to read STUFF.STM, create STUFF.OBJ, and use the name STUFF as the public symbolic name for the data block.

Then you would add the following lines of source to your program:

```
{$L STUFF.OBJ}
procedure Stuff; external;
```

This tells Turbo Pascal's linker to link STUFF.OBJ into the program, and to use the name Stuff to refer to what it linked in.

Finally, call the MemIdStream's Init method as follows:

```
M.Init(@Stuff, 2000);
```

This assigns the address of the Stuff procedure to the base of the IdStream region and provides the original size of the stream file as the size of this stream.

Once a memory stream is initialized in this manner, it can be used like any other stream. Note that the stream manager will generate an error if requested to read or write beyond the specified size of the stream.

Memory streams can be written to as well, although the need for this is less apparent.

Note that MemIdStream does not override the IdStream Done destructor, since there is no file to close and usually no memory buffer to deallocate.

Read / Truncate / Write

Declaration
```
procedure Read(var Buf; Count : Word); virtual;
```

Purpose
Read bytes from memory stream.

Description
Copies Count bytes from the current memory position to the variable Buf. If Count bytes would extend beyond the originally specified size of the stream, Read fills Buf with Count zeros and generates an error (epNonFatal+ecDiskRead)--it does not return a partial buffer.

See Also
DosIdStream.Read

Declaration
```
procedure Truncate; virtual;
```

Purpose
Truncate at current position.

Description
Reduces the effective size of the memory stream. This effect cannot be reversed in a MemIdStream except by calling Done and then Init again.

Declaration
```
procedure Write(var Buf; Count : Word); virtual;
```

Purpose
Write bytes to memory stream.

Description
Copies Count bytes from Buf to the current position in the memory stream. If fewer than Count bytes remain at the end of the memory stream, Write generates an error (epNonFatal+ecDiskFull) and writes nothing--it does not perform a partial write.

See Also
DosIdStream.Write

7. Libraries

A Library is a specialized form of stream: one that provides the ability to access individual objects randomly by name rather than by reading and writing them sequentially. The Library is used to great advantage by **Object Professional's** MAKEMENU and MAKESCRN utilities. There, library files store menu systems and/or data entry screens, each of which can be selected for further modification by using a pick list.

The physical storage format of a library is similar to that of the LU format designed originally for CP/M. A library file begins with a directory structure of a predetermined size. Each directory entry contains an ASCII name (up to 12 characters long) used to identify the entry, the file position and length of the object image, the type code and Load version number of the object, flags indicating the status of the entry, and perhaps other user-defined information. The DirEntry object manages each directory entry. A Library object contains a pointer to an array of pointers to DirEntry objects. Pointers are used because each item can vary in size and the number of items can vary. While the Library is open, the directory is maintained simultaneously in memory and on disk.

Following the library directory are the object images; these extend to an indefinite length. Because library entries may be deleted, some of the object images may be obsolete. Deleted images consume space until the library is packed and the space reclaimed.

Note that the Library object does not produce files that are directly compatible with LU file utilities. Library files can be accessed only via the methods of the Library object itself.

Because the Library is itself defined as an object, it can be extended. For example, an enhanced library type might allow multiple revisions of a given object within the same library.

Any object that can be stored in a stream can be stored in a library. Like a stream, a single library can store more than one object type. For example, one library file can hold both data entry screens and menu systems. The techniques for registering data types in a library are the same as those for a stream. The Library object is derived from BufIdStream and inherits all the properties of that stream type.

The following short program demonstrates the major features of the Library type.

```
program LibraryExample;  {EXLIB.PAS}
uses OpRoot;
const
  otNameData = 1000;          {Name object type}
  veNameData = 0;             {Name object version}
type
  {An object for manipulating Name and Age}
  NameDataPtr = ^NameData;
  NameData =
    object(Root)
      Name : String[20];
      Age : Byte;
      constructor Init;
      procedure Display;
      constructor Load(var S : IdStream);
      procedure Store(var S : IdStream);
    end;
var
  Name : NameData;
  Lib : Library;
  I : Integer;
```

```
  S : String[3];
  Status : Word;

  {----- NameData Methods ------------------}
  constructor NameData.Init;
  begin
    WriteLn;
    Write('Name:  ');
    ReadLn(Name);
    Write('Age:   ');
    ReadLn(Age);
  end;

  procedure NameData.Display;
  begin
    WriteLn;
    Writeln(Name, ' ', Age);
  end;

  constructor NameData.Load(var S : IdStream);
  begin
    if not Root.Init then Fail;
    Name := S.ReadString;
    S.Read(Age, SizeOf(Age));
    if S.PeekStatus <> 0 then Fail;
  end;

  procedure NameData.Store(var S : IdStream);
  begin
    S.WriteString(Name);
    S.Write(Age, SizeOf(Age));
  end;

  {$F+}
  procedure NameDataStream(SPtr : IdStreamPtr);
  begin
    SPtr^.RegisterType(otNameData, veNameData, TypeOf(NameData),
                       @NameData.Store, @NameData.Load);
  end;
  {$F-}

begin
  {Create a Library and register the NameData object}
  if not Lib.Create('NAMEDATA.OPL', 1024, 'NAMEDATA', 25) then begin
    WriteLn('Failed to create Library. Status = ', InitStatus); Halt;
  end;
  Lib.RegisterHier(NameDataStream);

  {Create and Store three NameData objects}
  for I := 1 to 3 do begin
    Name.Init;
    Str(I, S);
    Lib.PutEntry('NAME'+S, Name);
    Status := Lib.GetStatus;
    if Status <> 0 then
      WriteLn('Failed to Add NAME', S, ', Status = ', Status)
    else
      WriteLn('NAME', S, ' added to Library');
    Name.Done;
  end;
  Lib.Done;
```

```
  {Re-open the existing Library, Re-register the objects}
  if not Lib.Init('NAMEDATA.OPL', SOpen, 1024, 'NAMEDATA') then begin
    WriteLn('Failed to Open Library,  Status = ', InitStatus);
    Halt;
  end;
  Lib.RegisterHier(NameDataStream);

  {Retrieve the third entry by name, display it, then close the Library}
  Lib.GetEntry('NAME3', Name);
  if Lib.GetStatus <> 0 then begin
    WriteLn('Error reading NAME3');
    Halt;
  end;
  Name.Display;
  Lib.Done;
  Name.Done;
end.
```

The NameData object here is identical to the one used to demonstrate the stream objects. Libraries use the same concepts of type registration and the same Load and Store methods as do streams.

Create is one of two constructors provided by the Library object. Create makes a new, empty library file. Its first parameter is the DOS filename of the Library. (By convention, we use the extension OPL--Object Professional Library--to indicate a library file.) The second parameter is the buffer size used by BufIdStream to improve performance while reading or writing an object in the library. The third parameter is a string, up to 8 characters long, which is stored at the start of the library file to identify it as a valid library. The final parameter is the number of directory entries to reserve for the library. In this example, the library can store up to 25 unique objects at a time.

The Create call can fail if an invalid filename is specified, if there are no file handles available, if insufficient heap space is available for the buffer and the in-memory directory structures, or if a disk write error occurs.

Object types to be stored in the library must be registered using the RegisterHier or RegisterType methods. The example library stores only one kind of object, the NameData type, but generally libraries may store any number of object types. As with streams, the object types stored in a Library must always be derived from the Root object.

The PutEntry method either adds a new entry to the library or replaces an existing one. In the example, we add three instances of the NameData object to the library, naming them NAME1, NAME2, and NAME3. If one of these instances were replaced, the old image would remain in the library and the new one would be added to the end of the library file. The old image would be inaccessible since the directory entry would be updated to point at the new image only. The Library object provides a Pack method which squeezes out the inaccessible portions of the file.

The Done method flushes the Library's buffer, closes the DOS file, and deallocates the stream's registered type list.

The Init constructor of Library opens an existing library file. Its Parameters include the filename, the open mode (reading only, writing only, read/write), the size of the memory buffer, and the expected library signature. This constructor will fail if the file isn't found, if insufficient heap space is available, or if the expected signature isn't found.

After a successful call to Init, the stream's data types must again be registered with the stream manager.

The example demonstrates random access to the library by calling GetEntry to read the last element in the library, NAME3. GetEntry initializes the static object instance Name. A similar method, GetEntryPtr, is available to allocate and initialize a dynamic instance of an object. GetEntry will fail if no matching directory entry is found, if a disk error occurs, or if the Load constructor called to read the object image fails.

The MemLibrary object has the same relationship to the Library as the MemIdStream does to the BufIdStream. A MemLibrary is bound into a program so that a library can be accessed without the requirement for additional files. From an application's standpoint, most of the methods in MemLibrary work just like those in a Library. These methods will be documented only briefly.

The remainder of this section describes the methods associated with libraries. The following object types are covered:

```
[Root]
  DirEntry
[IdStream]
  [DosIdStream]
    [BufIdStream]
      Library
  [MemIdStream]
    MemLibrary
```

Constants

```
DirectorySig = Ord('O')+(Ord('P') shl 8); {'OP' => $504F}
```

Two byte signature that precedes each valid directory entry in a library.

```
TempLibExt : string[3] = '$$$';
```

Filename extension used when a library file is packed. See Library.Pack for more information.

```
deUnused  = $0001;
deDeleted = $0002;
```

Bit masks used to denote options in the DirEntry object. When the deUnused bit is set, the directory entry is not currently in use. When the deDeleted bit is set the entry describes an object that has been deleted. In this case, the object may still be recovered with the Library.UndeleteEntry method.

Types

```
DirEntryName = string[12];
```

A string of this type is used as the name of each library entry. The size is consistent with the traditional "eight-dot-three" format for DOS filenames. Library entry names are not constrained to DOS filename format, however.

```
DirEntryPtr = ^DirEntry;
DirEntry =
  object(Root)
    ...
    deUser       : Pointer;
    deCustom     : record end;
    ...
  end;
```

Object type used to manage each directory entry in a Library. The type was carefully designed to be extensible. Specifically, the data field deUser is set aside for your use at

runtime. (The SetUserPointer and GetUserPointer methods are used to access this field.) If a derived type adds new data fields, the curious field deCustom lets the directory manager read and write those new fields without even knowing what they contain.

```
DirEntries = array[1..512] of DirEntryPtr;
DirPtr = ^DirEntries;
```

Types that define an array of pointers to directory entries. The upper limit of 512 entries is arbitrary. The absolute upper limit of entries in a library would be 16380.

```
LibrarySig = string[8];
```

A string of this type is written at the start of every library file. It serves to identify the file as a valid library. Nine bytes are written no matter what the actual length of the string.

```
LibraryPtr = ^Library;
Library =
  object(BufIdStream)
    ...
  end;
```

Object used to manage a random-access file of object images.

```
MemLibraryPtr = ^MemLibrary;
MemLibrary =
  object(MemIdStream)
    ...
  end;
```

Object used to manage a random-access in-memory collection of object images.

deOptionsAreOn / deOptionsOff / deOptionsOn

Declaration

```
function deOptionsAreOn(Options : Word) : Boolean;
```

Purpose

Returns True if the specified flags are on.

Description

The only two option flags for the DirEntry object are deUnused and deDeleted. When deUnused is on, the directory entry is not currently in use. When deDeleted is on, the entry has been marked as deleted. Such an entry may be restored by using the Library.UndeleteEntry method.

This method is intended primarily for internal use by the Library object.

Example

```
var
  Lib : Library;
  DEP : DirEntryPtr;
begin
  ...
  DEP := Lib.FindAnyDirectoryEntry('NAME1');
  Write('Entry NAME1 is ');
  if DEP = nil then
    WriteLn('not found')
  else if DEP^.deOptionsAreOn(deDeleted) then
    WriteLn('deleted')
  else
    WriteLn('not deleted');
end.
```

Reports the status of the NAME1 entry.

Declaration

```
procedure deOptionsOff(Options : Word);
```

Purpose

Clear the specified flag(s).

Description

May be used to clear the option flags for the DirEntry object: deUnused and deDeleted.

This method is intended primarily for internal use by the Library object.

Declaration

```
procedure deOptionsOn(Options : Word);
```

Purpose

Set the specified flag(s).

Description

May be used to set the option flags for the DirEntry object: deUnused and deDeleted.

This method is intended primarily for internal use by the Library object.

Declaration

```
function GetEntryLength : LongInt;
```

Purpose

Return the length of the entry.

Description

The value returned is the number of bytes in the object image managed by the directory entry. In combination with GetEntryStart, this completely specifies the position and length of the object.

If the entry is currently unused, GetEntryLength returns 0.

Example

```
var
  Lib : Library;
  DEP : DirEntryPtr;
begin
  ...
  DEP := Lib.FindDirectoryEntry('NAME1');
  WriteLn('Entry length is ', DEP^.GetEntryLength);
end;
```

Reports the length of directory entry NAME1.

See Also

GetEntryStart

Declaration

```
function GetEntryName : DirEntryName;
```

Purpose

Return the name of the entry.

Description

If the entry is currently unused, returns the empty string. Otherwise, GetEntryName returns the name last assigned to the entry.

Example

```
WriteLn('First directory name is ', Lib.liDirectory^[1]^.GetEntryName);
```

This example directly references the directory pointer array in the Library object and reports the name of the first directory entry in that array.

See Also

Rename

GetEntryStart / GetUserPointer

Declaration

```
function GetEntryStart : LongInt;
```

Purpose

Return the starting point for the entry.

Description

The value returned is the byte offset into the library file for the object image managed by the directory entry. In combination with GetEntryLength, this completely specifies the position and length of the object.

If the entry is currently unused, GetEntryStart returns 0.

Example

```
var
  Lib : Library;
  DEP : DirEntryPtr;
begin
  ...
  DEP := Lib.FindDirectoryEntry('NAME1');
  WriteLn('Entry start location is ', DEP^.GetEntryStart);
end.
```

Reports the starting location of the NAME1 directory entry.

See Also

GetEntryLength

Declaration

```
function GetUserPointer : Pointer;
```

Purpose

Return the user pointer.

Description

Each directory entry reserves four bytes for user data storage. This area may be used for any purpose while the library is active in memory. For example, it might store a pointer to the actual instance of the object the entry refers to.

Note that this pointer value is *not* stored in the disk image of a library. That pointer is always set to nil.

Example

```
var
  Lib : Library;
  MyPtr : Pointer;
begin
  ...
  MyPtr := Lib.FindDirectoryEntry('NAME1')^.GetUserPointer;
end.
```

Returns the user pointer of the directory entry NAME1.

See Also

SetUserPointer

Declaration
```
constructor Init;
```

Purpose
Initialize an empty directory entry.

Description
Fills the directory entry with zeros, sets the signature to DirectorySig, and initializes the directory flags to indicate an unused entry.

This method is intended primarily for the internal use of the library manager.

See Also
Update

Declaration
```
procedure Rename(Name : DirEntryName);
```

Purpose
Change the name of the directory entry.

Description
May be used either to change the name of an existing entry or to provide a name for a previously unused entry.

See Also
Update

Declaration
```
procedure SetUserPointer(P : Pointer);
```

Purpose
Set the user pointer to P.

Description
Each directory entry reserves four bytes for user data storage. This area may be used for any purpose while the library is active in memory. For example, it might store a pointer to the actual instance of the object the entry refers to.

Note that this pointer value is *not* stored in the disk image of a library. That pointer is always set to nil.

Example
```
var
  Lib : Library;
  MyPtr : MyObjectPtr;
begin
  ...
  {Allocate and load NAME1 object from disk}
  MyPtr := Lib.GetEntryPtr('NAME1');
  {Store the object's address in its library directory}
  Lib.FindDirectoryEntry('NAME1')^.SetUserPointer(MyPtr);
end.
```

Allocate and load an object from a library, then store the address of the object in its directory entry.

See Also
GetUserPointer

Update

Declaration

```
procedure Update(Start, Len : LongInt; Code, Ver : Word); virtual;
```

Purpose

Update the object description.

Description

When an object is first written to a library, this method is called to store its position and length in the library file, and its type and version information for the stream manager. Later, if an entry is replaced, new values for Start and Len are assigned with the Update method.

This method is intended primarily for the internal use of the library manager.

Example

```
var
  Lib : Library;
  DEP : DirEntryPtr;
  NewStart, NewLen : LongInt;
begin
  ...
  DEP := Lib.FindDirectoryEntry('NAME1');
  DEP^.Update(NewStart, NewLen, DEP^.deCode, DEP^.deVer);
end.
```

Assigns new start and length values for the library entry NAME1.

See Also

Rename

Declaration

```
function AvailableEntries : Word;
```

Purpose

Return number of entries available in library's directory.

Description

The total number of entries for a given library is assigned when the library is created. AvailableEntries returns the remaining number of unused entries. It does not count deleted entries, which may be converted to unused by packing the library.

Example

```
WriteLn('Library Status:');
WriteLn('Maximum Entries:   ', Lib.MaxEntries);
WriteLn('Current Entries:   ', Lib.CurrentEntries);
WriteLn('Deleted Entries:   ', Lib.DeletedEntries);
WriteLn('Available Entries: ', Lib.AvailableEntries);
```

A library status display.

See Also

CurrentEntries MaxEntries
DeleteEntries

Declaration

```
constructor Create(FileName : PathStr; BufferSize : Word;
                   Sig : LibrarySig; Entries : Word);
```

Purpose

Create a new library.

Description

FileName is a valid DOS pathname for the library to create. BufferSize is the number of bytes of heap space to allocate for buffering input and output operations on the library. Sig is a string up to 8 bytes long that will serve to identify the file as a valid library. Entries is the number of unique directory entries reserved for the library. No more than this number of objects may be stored in the library.

Create allocates BufferSize+Entries*(4+SizeOf(DirEntry)) bytes of heap space. The size of a standard DirEntry is 35 bytes. Note that the library directory is maintained in memory. Hence, library directory searches are fast and require no disk access.

Create can fail if an invalid filename is specified, if there are no file handles available, if insufficient heap space is available for the buffer and the in-memory directory structures, or if a disk write error occurs. If Create fails, check the value of InitStatus to determine the cause of the error.

The library is left open and ready to write after calling Create. Before writing objects to the library, their types must be registered using the RegisterHier or RegisterType methods.

To open an existing library, use the Init constructor instead.

Example

```
if not Lib.Create('D:\NAMEDATA.OPL', 1024, 'NAMEDATA', 25) then begin
  WriteLn('Failed to create library. Status = ', InitStatus); Halt;
end;
Lib.RegisterHier(NameDataStream);
```

Create a library with the name D:\NAMEDATA.OPL and a signature of NAMEDATA. This library will have room for 25 entries and will use a 1K buffer for disk I/O.

See Also

IdStream.RegisterHier Init

CurrentEntries / DeletedEntries / DeleteEntry

Declaration
```
function CurrentEntries : Word;
```

Purpose
Return the number of used entries in the library's directory, not including deleted entries.

Description
The following equation describes the number of entries in a library:

```
MaxEntries = CurrentEntries+DeletedEntries+AvailableEntries
```

MaxEntries is the number of entries specified when the library was created.

See Also
AvailableEntries MaxEntries

Declaration
```
function DeletedEntries : Word;
```

Purpose
Return the number of deleted entries in the library's directory.

Description
Note that deleted entries may be undeleted by using the UndeleteEntry method. Deleted entries are converted to unused after a library is packed.

See Also
AvailableEntries MaxEntries

Declaration
```
procedure DeleteEntry(Name : DirEntryName); virtual;
```

Purpose
Mark the entry of the specified name as deleted.

Description
If Name isn't found in the library, an error (epNonFatal + ecEntryNotFound) is generated. Otherwise, the deDeleted flag is set in the directory entry and the library file is updated. Deleting an already deleted entry is not an error.

Undeletion of the same entry may be performed by calling UndeleteEntry.

Example
```
Lib.DeleteEntry('NAME1');
Status := Lib.GetStatus;
if Status <> 0 then
  Writeln('Entry not found, Status = ', Status);
```

Deletes the directory entry named NAME1.

See Also
Pack UndeleteEntry

Declaration

```
destructor Done; virtual;
```

Purpose

Flush and close library file, and dispose of directory and stream.

Description

Done deallocates all heap space used by the Library and closes the disk file.

Before your application halts be sure to call the Done method of any Library that has been written to. Otherwise, its contents will not be completely written to disk.

Example

```
Lib.Done;
```

Flushes the buffer, closes the library file and disposes of the library stream.

See Also

Create Init

Declaration

```
function FileNeedsPacking : Boolean;
```

Purpose

Returns True if library needs packing.

Description

A library needs packing if it contains one or more deleted entries or if one or more entries in the library have been updated (thus obsoleting an earlier version of an object). In either case, portions of the library disk file are just wasting space.

FileNeedsPacking works by comparing the physical size of the Library file to the size computed by summing the directory entries in use. As such, it is reasonably fast.

The Pack method calls FileNeedsPacking before doing anything else, so there is no need to make an explicit call to this method first. The primary need for FileNeedsPacking arises in displaying a library status screen such as those used by MAKESCRN and MAKEMENU.

Example

```
WriteLn('Needs Packing? ', Lib.FileNeedsPacking);
```

See Also

Pack PackedFileSize

FindAnyDirectoryEntry / FindDirectoryEntry

Declaration

```
function FindAnyDirectoryEntry(Name : DirEntryName) : DirEntryPtr;
```

Purpose

Return a pointer to a directory entry; nil if not found.

Description

FindAnyDirectoryEntry searches all entries, deleted or active. Comparison of the specified Name to each directory entry is not case sensitive. Note that if Name is the empty string, FindAnyDirectoryEntry will return a pointer to the first available entry.

If the pointer returned is not nil, it may be used to access the methods and data fields of the DirEntry object.

Example

```
var
  Lib : Library;
  DEP : DirEntryPtr;
begin
  ...
  Lib.DeleteEntry('NAME1');
  DEP := Lib.FindAnyDirectoryEntry('NAME1');
  if DEP = nil then
    WriteLn('Entry not found')
  else
    {...work with DEP...};
end.
```

DEP is assigned a pointer to the directory entry for NAME1 even though the entry was just deleted.

See Also

FindEntryByIndex

Declaration

```
function FindDirectoryEntry(Name : DirEntryName) : DirEntryPtr;
```

Purpose

Return a pointer to a directory entry; nil if not found or deleted.

Description

In contrast to FindAnyDirectoryEntry, this method does not search deleted entries. Comparison of the specified Name to each directory entry is not case sensitive.

If the pointer returned is not nil, it may be used to access the methods and data fields of the DirEntry object.

Example

See the example for FindAnyDirectoryEntry.

Declaration

```
function FindEntryByIndex(N : Word) : DirEntryPtr;
```

Purpose

Return a pointer to the N'th entry in the directory.

Description

Allows direct access to all the directory entries. Valid values of N are in the range 1..MaxEntries. If N is outside of this range, FindEntryByIndex returns nil. Otherwise the pointer returned may be used to access the methods and data fields of the DirEntry object.

Example

```
var
  Lib : Library;
  I : Word;
begin
  ...
  WriteLn('Entries in use:');
  for I := 1 to Lib.MaxEntries do
    with Lib.FindEntryByIndex(I)^ do
      if not deOptionsAreOn(deUnused) and not deOptionsAreOn(deDeleted) then
        WriteLn(I, ' ', GetEntryName);
end.
```

Writes a list of all active entries in the library.

See Also

FindAnyDirectoryEntry

Declaration

```
procedure GetCodeAndVersion(Name : DirEntryName; var Code, Ver : Word);
```

Purpose

Return the code and version for the specified entry.

Description

If Name is not found, GetCodeAndVersion generates an error (epNonFatal + ecEntryNotFound) and leaves the Code and Ver variables uninitialized. Otherwise, Code and Ver are set to the type code and version number of the object represented by Name.

This information may be used to determine whether the Load and Store methods for the object type have been registered with the stream manager. It may also be used to assure that the library element is consistent with an object instance variable to be passed to the GetEntry method.

Note that GetCodeAndVersion works for both active and deleted entries.

Example

```
var
  Code, Ver : Word;
...
  Lib.GetCodeAndVersion('NAME1', Code, Ver);
```

Looks up the code and version for the directory entry named NAME1.

See Also

IdStream.CodeVerRegistered

GetEntry / GetEntryPtr

Declaration

```
procedure GetEntry(Name : DirEntryName; var R : Root);
```

Purpose

Calls polymorphic Load constructor to read static object R.

Description

GetEntry first attempts to find the named entry in the library directory. If the search fails or if the entry has been deleted, it generates an error (epNonFatal + ecEntryNotFound). Otherwise, GetEntry uses the directory entry to seek to the appropriate location in the library file, then calls IdStream.Get to read the object into the parameter R.

GetEntry will also fail if a disk error occurs, if the data type is not registered, or if the Load constructor generates an error of its own. Use the GetStatus method to check for errors after calling GetEntry.

Be sure that R is a variable of a type large enough to hold the instance read by GetEntry. The compiler will not warn about an error of this type and the result will be corrupted memory. The GetCodeAndVersion method may be used to perform a runtime check in cases where the library holds more than one data type. Alternately, the GetEntryPtr method will automatically allocate a dynamic instance of the proper size no matter what the data type.

Example

```
Lib.GetEntry('NAME3', Name);
Status := Lib.GetStatus;
if Status <> 0 then
  Writeln('Failed to load NAME3, Status = ', Status);
```

GetEntry searches the directory for an entry with the name NAME3. If found, that object's Load constructor is called to read its data from disk.

See Also

GetCodeAndVersion
IdStream.GetStatus
PutEntry

Declaration

```
function GetEntryPtr(Name : DirEntryName) : RootPtr;
```

Purpose

Calls polymorphic Load constructor to allocate and read object.

Description

Works like GetEntry with the exception that it creates a dynamic instance rather than a static one. This means that heap space is allocated to hold the object and a pointer to that heap space is returned by GetEntryPtr. GetEntryPtr is preferred when the type of the object isn't known at compile time or when many instances are to be maintained in memory simultaneously.

GetEntryPtr returns nil if any error occurred during the call. Use the GetStatus method to determine an exact error code.

Example

```
NamePtr := NameDataPtr(Lib.GetEntryPtr('NAME3'));
if NamePtr = nil then
  Writeln('Failed to load NAME3, Status = ', Lib.GetStatus);
```

GetEntry searches the directory for an entry with the name NAME3. If found, that object's Load constructor is called to read its data from disk. The constructor will allocate the heap space required to hold the Name object.

Declaration

```
constructor Init(FileName : PathStr; Mode, BufferSize : Word; Sig : LibrarySig);
```

Purpose

Open an existing library.

Description

FileName is the pathname of an existing library to open. Mode specifies how to open the file (for reading, writing, or both). Use the constants (SOpenRead, SOpenWrite, SOpen) defined at the start of the Streams section for this purpose. BufferSize is the number of bytes of heap space to allocate for buffering input and output operations on the library. A value of 1024 usually provides a good balance between speed and memory usage. Sig is a string up to 8 bytes long that serves to identify the file as a valid library. The string specified must match that used when the library was created.

Init allocates BufferSize+Entries*(4+SizeOf(DirEntry)) bytes of heap space, where Entries is the number of directory entries specified when the library was created. The size of a standard DirEntry is 35 bytes. Note that the library directory is maintained in memory while the library is open. Hence, directory searches take very little time and require no disk access.

Init will fail if an invalid filename is specified, if there are no file handles available, if insufficient heap space is available for the buffer and directory structures, if a disk read error occurs, or if the file's signature doesn't match Sig. If Init fails, check the value of InitStatus to determine the cause of the error.

The library is left open and ready to access after calling Init. Before reading or writing objects, their types must be registered using the RegisterHier or RegisterType methods.

To create a new library, use the Create constructor instead.

Example

```
if not Lib.Init('NAMEDATA.OPL', SOpen, 1024, 'NAMEDATA') then begin
  WriteLn('Failed to Open Library,  Status = ', InitStatus);
  Halt;
end;
Lib.RegisterHier(NameDataStream);
```

Opens an existing library file (for reading and writing) and registers a data type that it is presumed to contain.

```
if not Lib.Init('NAMEDATA.OPL', SCreate, 1024, 'NAMEDATA') then begin
  ....
```

Incorrect! You can't use Init to create a new library. Use Create instead.

See Also

Create IdStream.RegisterHier

KillEntry / MaxEntries

Declaration

```
procedure KillEntry(Name : DirEntryName);
```

Purpose

Permanently delete the entry of the specified name.

Description

An entry deleted with KillEntry cannot be undeleted. The matching directory entry is reinitialized just like a new entry. Note that the data associated with the entry still remains in the library file (wasting space) until the library is packed.

Example

```
Lib.KillEntry('NAME1');
Status := Lib.GetStatus;
if Status <> 0 then
  Writeln('Entry not found, Status = ', Status);
```

Permanently removes the directory entry named NAME1.

```
Lib.KillEntry('NAME1');
Lib.UnDeleteEntry('NAME1');
```

Incorrect! KillEntry reinitialized the directory entry; NAME1 is completely gone.

See Also

DeleteEntry Pack

Declaration

```
function MaxEntries : Word;
```

Purpose

Return the maximum number of entries in the library's directory.

Description

MaxEntries is the same as the number of entries specified when the library was created.

Example

```
WriteLn('Library Status:');
WriteLn('Maximum Entries:   ', Lib.MaxEntries);
WriteLn('Current Entries:   ', Lib.CurrentEntries);
WriteLn('Deleted Entries:   ', Lib.DeletedEntries);
WriteLn('Available Entries: ', Lib.AvailableEntries);
```

A library status display.

See Also

AvailableEntries DeletedEntries
CurrentEntries

Declaration

```
procedure Pack;
```

Purpose

Pack the library.

Description

Pack removes wasted data space from a library. Wasted space is generated when entries are deleted or updated, in which case the old data remains in the library until it is packed. Pack first checks to see whether the library has any wasted space. If there is none, it exits immediately.

The library to be packed must have been initialized previously by calling either the Create or Init constructor. When the Init constructor was used, the library must have been opened in read/write mode (SOpen); otherwise, Pack will exit with an error.

Pack creates a temporary work file by forcing the extension TempLibExt ('$$$' by default) onto the name of the library being packed. The library's old directory is first copied to the work file. Deleted entries, however, are reinitialized before copying; hence, they cannot be undeleted after packing. Then each valid data object is copied from the old file to the new one using the library's memory buffer as a copy buffer. (No Load and Store methods are involved; this is strictly a byte-level copy.) If all of this succeeds, the original library is closed and erased, and the work file is renamed to the original library name. Finally, the library is reopened in its original mode.

Use the GetStatus method to check for errors. If an error is detected, the library's Done method should be called immediately. In almost all cases, the original library file will be left intact. In one case (if the final rename fails), the valid packed library file will still have the extension TempLibExt and the original library file will no longer exist.

Example

```
Lib.DeleteEntry('NAME3');
Lib.Pack;
Status := Lib.GetStatus;
if Status <> 0 then begin
  WriteLn('Pack Failed, Status = ', Status);
  Lib.Done;
  Halt;
end;
```

The request to delete NAME3 merely marks the directory entry as deleted. The call to Pack physically removes NAME3 by copying everything but NAME3 to a new file.

See Also

DeleteEntry PutEntry

PackedFileSize / PutEntry

Declaration

```
function PackedFileSize : LongInt;
```

Purpose

Returns size of library after packing.

Description

The Pack method needs at least PackedFileSize bytes of disk space free at the time packing begins, to use for a temporary work file. The FileNeedsPacking method compares PackedFileSize to the actual file size to determine whether packing is required.

Example

```
if Lib.FileNeedsPacking then
  if DiskFree(0) < Lib.PackedFileSize then
    WriteLn('Not enough disk space free to pack')
  else
    Lib.Pack;
```

Packs a library if it needs it and adequate disk space is available. This example assumes the file is on the current drive and calls the DOS unit's DiskFree method to get the free space.

See Also

Pack

Declaration

```
procedure PutEntry(Name : DirEntryName; var R : Root);
```

Purpose

Calls polymorphic Store method to write object R.

Description

PutEntry either creates a new directory entry with the specified name or else updates the existing entry of the same name. The specified Name may not be an empty string. Directory searching is not case sensitive; choose unique directory names accordingly.

PutEntry first finds an available directory entry. If no entry matches Name and no entries remain unused, PutEntry generates an error (epNonFatal + ecDirectoryFull) and exits. Next it seeks to the end of the library file and calls IdStream.Put to write the image of object R there. Put will fail if the object type has not been registered or if a disk error occurs. Finally, PutEntry updates the directory entry to contain the name, type, position, and length of the object just written.

If PutEntry is used to update the contents of an existing entry, it leaves wasted space behind in the library file. This space may be recovered by calling Pack.

Example

```
Init(Name);
Lib.PutEntry('NAME1', Name);
Status := Lib.GetStatus;
if Status <> 0 then
  Writeln('Failed to Add NAME1, Status = ', Status);
```

Creates an instance of the Name object and writes that instance (calling it NAME1) to the library. This creates a directory entry and writes the object to disk. As with any other stream, the object being written (Name) must have Root as its ultimate ancestor.

```
New(NamePtr, Init);
Lib.PutEntry('NAME1', NamePtr^);
```

Writes a dynamic instance to the library.

See Also

GetEntry IdStream.Put

Declaration

```
procedure RenameEntry(OldName, NewName : DirEntryName); virtual;
```

Purpose

Rename the specified entry.

Description

The directory entry for OldName is modified to specify NewName instead. An error will occur if NewName matches another existing entry or if OldName is not found in the library.

RenameEntry does not generate wasted space in the library.

Example

```
Lib.RenameEntry('NAME1', 'NAME99');
```

Renames the entry NAME1 to NAME99.

Declaration

```
procedure UndeleteEntry(Name : DirEntryName); virtual;
```

Purpose

Remove deletion mark from the specified entry.

Description

Use UndeleteEntry to undo a call to DeleteEntry. This is possible only if the library has not been packed since the call to DeleteEntry. Closing and reopening a library (without packing it) does not affect the ability to undelete an entry.

Undeleting an entry that was never deleted is not an error.

To delete an entry so that it is impossible to recover, either call KillEntry or else call DeleteEntry and then Pack the library.

Example

```
Lib.UnDeleteEntry('NAME1');
Status := Lib.GetStatus;
if Status <> 0 then
  Writeln('Entry not found, Status = ', Status);
```

Undeletes the directory entry named NAME1.

See Also

DeleteEntry

GetEntry / GetEntryPtr

Declaration

```
procedure GetEntry(Name : DirEntryName; var R : Root);
```

Purpose

Calls polymorphic Load constructor to read static object R.

Description

Works just like Library.GetEntry, with the exception that there is no disk access: the object is transferred directly from one memory location to another.

Note that there is no PutEntry method for a MemLibrary. A MemLibrary is assumed to be created originally as a disk file and then converted to a linkable memory image for read-only use. See the Init method for further information.

Declaration

```
function GetEntryPtr(Name : DirEntryName) : RootPtr;
```

Purpose

Calls polymorphic Load constructor to allocate and read object.

Description

Works just like Library.GetEntryPtr, with the exception that there is no disk access: heap space for the new object is allocated and the space is initialized by copying directly from the library's memory location.

Declaration

```
constructor Init(BasePtr : Pointer; Size : Word);
```

Purpose

Open an existing library in memory.

Description

A MemLibrary is created by a multi-step process that starts with a normal Library object. The Library object may be used to add and modify entries as desired. If updates are performed, the Library should be packed as a final step.

Next, Borland's BINOBJ utility will convert the library file to a linkable OBJ file. Assuming a library file called NAMEDATA.OPL, the following command could be entered at the DOS prompt:

```
BINOBJ NAMEDATA.OPL NAMEDATA.OBJ NAMELIB
```

The final parameter supplies a symbolic name (NAMELIB) by which an application can refer to the contents of the OBJ file.

The OBJ file is linked into an application program and used as follows:

```
const
  NameLibSize = 5000;        {must be >= actual size}
var
  Lib : MemLibrary;

  {$L NAMEDATA.OBJ}
  procedure NameLib; external;

begin
  {Open the bound-in library}
  if not Lib.Init(@NameLib, NameLibSize) then begin
    WriteLn('Failed to Open Library,  Status = ', InitStatus);
    Halt;
  end;
  ...
end.
```

The Size parameter passed to Init must be at least as large as the actual file size of the library's original OPL file. If it is too small, MemLibrary read operations will fail when they exceed the specified size.

The NameLib procedure isn't really executable code; it just provides a way to refer to the address of the NAMEDATA.OBJ data linked into the program via the $L compiler directive.

We pass the address of NameLib to the Init constructor using the Turbo Pascal '@' operator.

Once the MemLibrary has been initialized successfully, it can be read just like a normal disk-based Library object. Note that registration of data types is still required.

When an application is finished with a MemLibrary, it should call the Done method to assure that the IdStream on which it depends is deallocated.

See Also

IdStream.RegisterHier

8. Error Handling

Object Professional implements a consistent method of error handling for all of its object types. The methodology includes a standardized set of numeric error codes, a shared set of error message strings, and a common error handling procedure that is called when an error is detected.

Error codes are stored in variables of type Word. Each code is computed as the sum of two factors: an error prefix, which is always a multiple of 10000; and an error number, which is always less than 10000. An error "type" is defined as the error prefix divided by 10000. Hence, given an error Code, you can determine its type and number with the following statements:

```
ErrorType   := Code div 10000;
ErrorNumber := Code mod 10000;
```

or, putting the Code back together again:

```
Code := (10000 * ErrorType) + ErrorNumber;
```

OPROOT defines five categories of error types. In descending order of severity, they are: fatal, non-fatal, warning, message, and non-error. By definition, when a fatal error occurs, the affected object is no longer usable. Note that it isn't essential for the entire application to halt when a fatal error occurs, although if the object is critical enough that may be the only recourse. All errors that occur in an object's constructor are obviously fatal. Common examples of fatal errors occur when there isn't enough memory to allocate a required buffer or when a required file cannot be opened.

A non-fatal error means that the requested operation did not occur, but the object remains usable. Examples of such errors include requests to resize a zoomed window, to load a non-existent file into an editor, to display help for a non-existent topic, and so on.

Warnings are typically associated with invalid user input in response to a prompt. For example, the field editors in the data entry system report a warning when the validation routine fails. Other examples of warnings: the search string specified for an editor isn't found; there are more files in a directory than will fit into the OPDIR data buffers; an attempt is made to select from a pick list that has no items.

At this time, the message category is used only by the OPBROWSE and OPEDITOR units when it passes strings to user-defined routines that prompt for a file to read or write, a search string, search options, or a line number. The specific message code and message string are consistent with, and in some cases are shared in common with, the other error codes and strings.

Finally, the non-error category is reserved to mean that no error has occurred. **Object Professional** makes no use of this category.

The other part of an error code, the error number, provides more detail about the error. The lowest error numbers, ranging from 2 (file not found) to 162 (general failure), are the same as Turbo Pascal's runtime error codes. Other numbers are organized loosely in blocks of 1000, with each block indicating another category of error:

```
0000-0999   DOS, Turbo Pascal, DOS critical errors
1000-1999   Other I/O errors, including stream-related errors
2000-2999   Capacity or environment errors
7000-7999   Warnings, user errors
8000-8999   Programming errors
9000-9999   Messages
```

A detailed listing of all such error codes is provided in Appendix D.

For the most common errors and messages, OPROOT provides a descriptive string that can be displayed by the associated error handler. All of these strings are declared in the file OPROOT.IN1. If you need to translate an application to another language, almost all of the changes you'll need to make will be found in this file. You'll note that OPROOT.IN1 precedes each string with its own CONST keyword. This is a signal to the Turbo Pascal linker for it to remove all the strings that aren't actually referenced by your program.

There are three different ways that your program may be informed of error codes that occur in object-oriented applications:

- constructor errors, reported in InitStatus
- errors returned by a function method such as CommandWindow.GetLastError
- errors reported to a user-written error handling routine

OPROOT declares a global typed constant named InitStatus, of type Word. Whenever an object's constructor fails, it returns False or a nil pointer, depending on whether a static or dynamic instance is being created. If your program detects failure, it should immediately check the value of InitStatus, where the constructor will have stored a standard error code indicating the reason for the failure. If no error has occurred, InitStatus will contain zero.

Most **Object Professional** objects provide a data field to store the most recent error code and a method to retrieve the value of that field. Exactly how this is implemented depends on the object. In all cases, you can check the error code to assure the success of other method calls. See the following error-related methods for more details:

OPROOT
IdStream.GetStatus
PointerStack.GetStatus
StaticQueue.GetStatus
LargeBitSet.GetStatus
StringDict.GetStatus

OPLARRAY
AbstractArray.ErrorA

OPCLONE
Cloner.GetLastError

OPCMD
CommandProcessor.GetLastError

OPEDIT
LineEditor.GetLastError

OPFRAME
Frame.GetLastError

OPWINDOW
RawWindow.GetLastError
CommandWindow.GetLastError

OPPRNLOW
BasePrinter.PrnError

OPFORM
PrintedForm.fGetLastError
Report.rGetLastError

Note that all the objects derived from CommandWindow inherit its GetLastError method.

Finally, many of the objects allow you to install a user-written error handler: a routine that the object itself calls when it detects an error. When you install such a routine you avoid having to call the GetLastError function since the responsibilities are reversed.

OPROOT declares the procedure type ErrorProc, which defines the parameters passed to the user-written routine. See the discussion of the "Error Handling Hook" in OPWINDOW's CommandWindow section for more discussion of this capability. A default error procedure, DefaultErrorProc, is defined in the OPCMD unit. Also see the following error-related methods in other units:

OPLARRAY
AbstractArray.SetErrorProc

OPEDIT
LineEditor.SetErrorProc

OPWINDOW
CommandWindow.SetErrorProc

The error handler we've been describing is implemented by using procedure variables. A single error handling procedure, by interpreting the unit and error codes passed to it, can handle more than one object type. Another error handling technique, more in line with OOP principles but perhaps less efficient, is also available.

Many of the objects include a virtual method that they call to report an error whenever one occurs. From each such object, you can derive a new object type that overrides the virtual method to provide different error handling behavior. The objects shown above with GetStatus and GetLastError methods have such virtual methods, typically named Error or GotError.

Constants

```
ucNone           = 00; {no unit}
ucEdit           = 01; {OPEDIT}
ucPick           = 02; {OPPICK}
ucDir            = 03; {OPDIR}
ucMenu           = 04; {OPMENU}
ucSelect         = 05; {OPSELECT}
ucEntry          = 06; {OPENTRY}
ucMemo           = 07; {OPMEMO}
ucEditor         = 08; {OPEDITOR}
ucBrowse         = 09; {OPBROWSE}
ucLArray         = 10; {OPLARRAY}
ucMacEd          = 11; {OPMACED}
ucHelp           = 12; {OPHELP}
ucSEdit          = 13; {OPSEDIT}
ucCalc           = 25; {OPCALC}
ucQkRef          = 26; {OPQKREF}
ucCal            = 27; {OPCAL}
ucReservedForTP  = 99;
```

Unique code numbers for each unit in **Object Professional** that supports a standard error or help handler. Such a code is passed as a parameter to the handler, allowing a program to

detect where the error or help request occurred. TurboPower reserves the codes 0..99 for its own use. The range 100..255 is available for your programs. See the ErrorProc type in this section for more information.

```
etFatal          = 0;  {fatal errors}
etNonFatal       = 1;  {non-fatal I/O errors}
etWarning        = 2;  {warnings, user input errors, etc.}
etMessage        = 3;  {simple messages}
etNoError        = 4;  {no error}
```

Error type codes used to classify errors.

```
epFatal          = etFatal * 10000;
epNonFatal       = etNonFatal * 10000;
epWarning        = etWarning * 10000;
epMessage        = etMessage * 10000;
```

Error prefixes added to error numbers before reporting them.

```
ecFileNotFound   = 00002; {file not found}
...
mcReplaceYesNo   = 09012; {prompt to confirm a replacement}
```

Error numbers. See Appendix D for a complete list.

```
emNoMoreFiles : string[17] = 'No matching files';
...
emDiagNotAllowed : string[26] = 'Diagonal lines not allowed';
```

Error message strings passed to user-written error procedures of type ErrorProc. Not all error numbers have an associated message string, but the most common ones do. See OPROOT.IN1 for a complete list of all the strings.

```
InitStatus : Word = 0;
```

Global variable used by all **Object Professional** constructors to store a status code when the constructor fails. The constructor itself will return a value of False or a nil pointer (for dynamic instances) to indicate failure. When this occurs, the program can check InitStatus for further details about the cause of the failure. Values stored in InitStatus use the standard **Object Professional** error codes described here.

Types

```
ErrorProc = procedure (UnitCode : Byte; var ErrorCode : Word;
                       ErrorMsg : string);
```

Procedure type that defines a standard error reporting mechanism for many units in **Object Professional**. See the unit codes above for a list of the units and Appendix D for a complete list of error codes.

C. OPLARRAY: Object Oriented Large Arrays

Although Turbo Pascal provides a large code model, and has always offered access to all 640K of DOS memory via the heap, it still does not allow individual data items to exceed 64K bytes in size. **Object Professional**'s solution to this limitation is the OPLARRAY unit. **Object Professional**'s large arrays can be one- or two-dimensional, and array elements can be of any Turbo Pascal type.

The large array unit allows the creation of arrays of virtually any size. The arrays may be stored in normal RAM, in Lotus/Intel/Microsoft expanded memory (EMS), or paged virtually to disk. Large arrays in normal RAM are stored on the Pascal heap, and as a result are limited to such a size as will fit into the available portion of the heap. Large arrays stored in EMS are limited to the amount of EMS memory available. Large arrays paged to disk are limited primarily by disk space and the maximum file size of the operating system (typically 32 megabytes for DOS).

OPLARRAY defines several object types for managing large arrays, arranged in the following hierarchy;

```
AbstractArray
  RamArray
    RamRCArray
  EmsArray
    EmsRCArray
  VirtualArray
    PrimitiveArray
    VirtualRCArray
  OpArray
```

AbstractArray serves to define the methods provided by all of the array types: how to store an element, retrieve it, and so on. RamArray, EmsArray, and VirtualArray implement the storage mechanism for heap-based, EMS-based, and disk-based arrays, respectively. RamRCArray, EmsRCArray, and VirtualRCArray add range-checking capabilities, which are useful during development of a large array application. By creating separate objects for range checking, the production versions of the objects avoid any performance overhead. PrimitiveArray is a variety of VirtualArray that allows you to specify the exact paging properties, a feature that can improve performance substantially for applications that have well-defined array access patterns. OpArray (short for Optimized Array), can store the array using any one of the three storage mechanisms: RAM, EMS, or disk. It chooses a best fit at runtime based on a search priority that you provide. In this way a program can optimize the performance of large arrays without knowing in advance what system resources will be available.

With the primary exception of the constructors, all of the large array objects implement the same methods. As a result, the method section of this chapter describes all the objects at the same time, pointing out differences only as necessary.

The large arrays implemented by the OPLARRAY unit are fully dynamic--that is, the dimensions and data type of arrays may be specified at runtime rather than at compile time. As a result, a single copy of the large array code will manage multiple data arrays of differing sizes and types in a single program. The disadvantage of this approach is that the large array unit cannot define a function that returns the value of an array element, since the type of the element is not known until runtime. Instead, each array object provides a method that returns array elements as untyped var parameters. Your application can easily define a shell function that calls this method and returns a function result of the appropriate type. Actually, this is an ideal opportunity for you to take advantage of OOP: derive a new object type with a

constructor specialized for your array type, a procedure method to assign values to the array, and a function method to retrieve them.

Arrays managed by **Object Professional**'s large array units are always zero-based: the minimum index on each axis of an array is always zero. When a new array is allocated, you will specify the total number of rows and columns of the array. When the array is accessed, the highest array index will be one less than the actual number of rows or columns.

Here's a small example program that uses the OpArray object type:

```
program OpArrayExample;  {EXLARRAY.PAS}
uses
  OpRoot, OpLarray;
type
  Str20 = String[20];
  Str20Array =
    object(OpArray)
      constructor Init(Rows : Word; FileName : String; HeapToUse : LongInt);
      procedure SetVal(Row : Word; Val : Str20);
      function GetVal(Row : Word) : Str20;
    end;

constructor Str20Array.Init(Rows : Word; FileName : String;
                            HeapToUse : LongInt);
const
  Priority : AutoPriority = (lRamArray, lEmsArray, lVirtualArray);
var
  ClearEl : Str20;
begin
  if not OpArray.Init(Rows, 1,          {Rows, columns}
                      SizeOf(Str20),    {Size of one element}
                      FileName,         {Virtual array file}
                      HeapToUse,        {Heap space to use}
                      0,                {Array options}
                      Priority)         {Allocation priority}
  then
     Fail;
  {Clear the array quickly}
  ClearEl := '';
  ClearA(ClearEl, FastInit);
end;

procedure Str20Array.SetVal(Row : Word; Val : Str20);
begin
  SetA(Row, 0, Val);
end;

function Str20Array.GetVal(Row : Word) : Str20;
var
  S : Str20;
begin
  RetA(Row, 0, S);
  GetVal := S;
end;

var
  OA : Str20Array;
```

```
begin
  if not OA.Init(1000,                {Rows}
                 'TEMP.ARR',          {Virtual array file}
                 MaxAvail div 2)      {Heap space to use}
  then begin
    WriteLn('Failed to init Str20Array. Status = ', InitStatus);
    Halt;
  end;

  case OA.TypeOfArray of
    lRamArray : WriteLn('RAM array');
    lEmsArray : WriteLn('EMS array');
    lVirtualArray : WriteLn('Virtual array');
  end;

  {Set the first element of the array}
  OA.SetVal(0, 'Object');

  {Write out the element}
  Writeln('OA[0] = ', OA.GetVal(0));

  {Dispose of the array}
  OA.Done;
end.
```

This example demonstrates the object-oriented approach to making large arrays of a specific type. It declares a new object type, Str20Array, which is derived from the OpArray. Str20Array inherits all of OpArray's behavior, including its ability to select the array's storage location based on a priority list. Str20Array needs to implement only three methods. Its constructor hides some of OpArray's details since it knows already that the array is one-dimensional, has elements of type Str20, and uses the specified priority list. The constructor also clears the array, a step that the OPLARRAY constructors do not take because it would be a waste of time for many applications. Str20Array's SetVal method takes only a single Row index and a value parameter of type Str20. And its GetVal method can be a function since it will always return an element of type Str20.

The main block instantiates an Str20Array with 1000 elements, writes out the actual storage type chosen for the array, stores a value in the first element, and retrieves that element again. You may wish to experiment by rearranging the order of the Priority constant.

Large Array Performance

Data types exceeding 64K bytes inherently take longer to access than smaller objects when using the Intel 8086 family architecture. Furthermore, dynamically allocated structures take longer to access than statically addressed ones. For each access to an array element, **Object Professional**'s routines must calculate a page (segment) and offset within the page where the data actually will be found. This calculation is performed in assembly language to attain the best possible speed. The virtual and EMS arrays require an additional operation to assure that the desired page is loaded into normal memory.

The following table provides benchmark results for reading and writing elements in each array model. Turbo Pascal 5.5's performance for a small array is also included for comparison. Measurements were taken on an 8 MHz 80286 system. The benchmark program used a static instance of the OpArray object type, similar to that of the example above. (Thanks to Turbo Pascal 5.5's efficient VMT mechanism for managing virtual method calls, the OpArray object has no more overhead than any of the other large array objects.) Results are reported as the number of microseconds to read or write a single array element of the specified type.

Element	Turbo	RAM	Virtual	EMS
Byte	8	86	123	133
Word	8	85	123	133
Real	16	91	127	140
String[80]	96	150	186	229

The virtual model results do not include time for swapping from disk, nor do the EMS results include time for swapping page windows. Disk swap times will vary significantly depending on the hardware and the array's page organization. EMS page swaps take 100-150 microseconds.

Those familiar with the large array units from Turbo Professional will note that the performance of the OPLARRAY unit is slightly better than the performance of the TPRARRAY, TPVARRAY and TPEARRAY units.

Turbo Pascal allows an object to be declared as either a dynamic instance (the object's data is stored on the heap and referenced by a pointer) or a static instance (the object's data is stored in the data segment). The actual amount of data stored in the data segment by a static large array object is quite small (since the bulk of the data is stored on the heap or in EMS memory anyway), so there is little advantage to making large array objects dynamic. In fact, the extra level of indirection required to dereference a dynamic instance's pointer variable degrades performance.

Files Containing Large Arrays

All of the large array types can store and reload arrays from disk. The array files written by each of the large array types are compatible. For example, a file stored by a RamArray object can be read by a VirtualArray object. Arrays are read by calling the LoadA constructor and written by calling the StoreA method.

The LoadA constructor of the large array objects can read files created by the StoreA procedure of TPVARRAY, or the StoreVA procedure of TPRARRAY, from Turbo Professional 4.0 or 5.0.

Array Priorities

As mentioned above, the OpArray object dynamically determines the best place to store its data. It does so based on a priority list, which is an ordered array of storage locations. OpArray constructors allocate the array by trying each location specified by the priority list and stopping at the first location with enough free capacity to hold the array.

The priority list is an array with three elements of type ArrayType. ArrayType is an enumerated type that defines the following values: lNoArray, lRamArray, lEmsArray, or lVirtualArray. OpArray will ignore list positions that contain lNoArray. By using lNoArray, arrays can be constrained to use only two of the three possible storage locations. A likely use of lNoArray is shown in the following example:

```
const
  SmallHeapPriorities : AutoPriority = (lRamArray, lEmsArray, lNoArray);
var LargeArray : OpArray;
begin
  if not LargeArray.Init(MaxRows, MaxCols, ElementSize,
                         ArrayFileName, HeapToUse, 0,
                         SmallHeapPriorities) then begin
    WriteLn('Failed to Init OpArray, Status = ', InitStatus);
    Halt;
  end;
  ...
```

SmallHeapPriorities specifies to try RAM storage first, then EMS storage. If neither location has space, the constructor will fail without ever trying disk storage, which is presumably too slow for this application.

OpArray provides two constructors: Init and LoadA. Init creates a new array. LoadA loads an existing array from a file. Like Init, LoadA steps through a specified priority list to determine where the array will be loaded.

One disadvantage of OpArray is that it causes the compiler to link in the code for all three storage mechanisms: RAM, EMS, and disk. Because the mechanism to use isn't determined until run-time, the compiler has no choice. OpArray clearly shows the advantages and disadvantages of OOP's late binding mechanism.

Error Handling

ErrorA is a method that returns the status of the most recent array operation. A call to ErrorA returns a Word error code and resets the internal status variable to 0. The codes returned by ErrorA are consistent with those used by the rest of **Object Professional**; see Appendix D for details.

Alternatively, applications can use the standard error handling hook provided by OPLARRAY. To do so, the application passes a procedure of type ErrorProc (defined in OPROOT) to the OPLARRAY method SetErrorProc. Then, whenever an error occurs, OPLARRAY will call the specified procedure to report the error. See SetErrorProc for more information.

Range Checking

Just as Turbo Pascal can capture attempts to access an invalid element of a normal array, OPLARRAY can do so for any of its large array models. While Turbo Pascal range checking is activated with a compiler option ({$R+}), OPLARRAY range checking is enabled by declaring the arrays to be of a special derived object type: RamRCArray, EmsRCArray, or VirtualRCArray. When an invalid index is passed to a method of any of these objects, the method stores an error code accessible by ErrorA and calls the user-defined error handler, if any. The requested array operation is not completed.

Range checking prevents out of range array indices from corrupting memory. Therefore, it is recommended that you declare array variables using the range-checked objects while designing and debugging programs. Since range checking does degrade performance of the large array, you may wish to switch to non-range-checked array variables once the program is debugged.

Range checking for the OpArray object is activated differently than for the other types. Do so by adding lRangeCheck to the ArrayOptions parameter passed to its constructor.

Common Mistakes

The single most common mistake made by large array users is to forget that array indices are always zero-based. In the most common case, the user has declared a one-dimensional array and tried to refer to the n'th element as row n, column 1, rather than row n-1, column 0. Please verify that you have not made such a mistake before calling our technical support line for assistance.

Constants

```
DefaultPriority  : AutoPriority = (lRamArray, lEMSArray, lVirtualArray);
```

Use this constant to initialize an OpArray object with the best possible performance given available system resources.

```
DefBeepOnError    : Boolean = True;
```

Determines whether the built-in error handlers will beep on an error. Affects both the default error handler and DisplayLErrorProc.

```
IdString          : string[6] = 'LARRAY';
```

The identification string written at the beginning of large array files created by OPLARRAY.

```
lKeepDiskCurrent = $0001;
lDeleteFile      = $0002;
lRangeCheck      = $0004;
```

These are the options that affect the behavior of a large array object. If lKeepDiskCurrent is enabled, the file associated with a virtual array is updated after each modification of the array. This leads to severely degraded performance, but prevents data loss in case of interrupted power or a program crash. When lDeleteFile is set, the Done destructor of a virtual array deletes the disk file as well as disposing of heap space. lRangeCheck determines whether an OpArray object performs range-checking. This flag is ignored for all other OPLARRAY objects.

```
MaxHeapBlock    = 65519;
```

Represents the largest manageable block of heap space, less one.

```
VArrayID          : String[6] = 'VARRAY';
```

The identification string written at the beginning of large array files created by TPVARRAY (from Turbo Professional).

Types

```
AbstractArrayPtr = ^AbstractArray;
AbstractArray =
  object(Root)
    ...
  end;
```

An abstract large array type from which all others are derived.

```
ArrayType         = (lNoArray, lRamArray, lEMSArray, lVirtualArray);
```

Enumerates the categories of large arrays. lNoArray is a special value described in "Array Priorities" above.

```
AutoPriority      = Array[1..3] of ArrayType;
```

Defines a priority list for the OpArray type. See the discussion of "Array Priorities" above.

```
EmsArrayPtr = ^EmsArray;
EmsArray =
  object(AbstractArray)
    ...
  end;
```

The EmsArray object, which stores the large array in Lotus/Intel/Microsoft Expanded Memory (EMS).

```
EmsRCArrayPtr = ^EmsRCArray;
EmsRCArray =
  object(EmsArray)
    ...
  end;
```

The EmsRCArray object, which implements range checking of EmsArray indices.

```
InitType          = (ExactInit, FastInit);
```

Defines initialization options for arrays. Passing the ExactInit value to ClearA causes that method to initialize each array element to an exact value of the proper type. Passing FastInit causes ClearA to set every byte of the array to the first byte of the initialization value. As its name implies, the second option is much faster. FastInit is useful for initializing numeric arrays to zero or string arrays to empty strings.

```
OpArrayPtr = ^OpArray;
OpArray =
  object(AbstractArray)
    ...
  end;
```

The OpArray object, which adapts its behavior to any of the other object types (with the exception of PrimitiveArray) based on available system resources.

```
PrimitiveArrayPtr = ^PrimitiveArray;
PrimitiveArray =
  object(VirtualArray)
    ...
  end;
```

The PrimitiveArray object, which works like a VirtualArray but provides full control over the internal page structure of the array.

```
RamArrayPtr = ^RamArray;
RamArray =
  object(AbstractArray)
    ...
  end;
```

The RamArray object, which stores the large array entirely on the Turbo Pascal heap.

```
RamRCArrayPtr = ^RamRCArray;
RamRCArray =
  object(RamArray)
    ...
  end;
```

The RamRCArray object, which implements range checking of RamArray indices.

```
VirtualArrayPtr = ^VirtualArray;
VirtualArray =
  object(AbstractArray)
    ...
  end;
```

The VirtualArray object, which pages the large array between a disk file and a Pascal heap buffer as needed.

```
VirtualRCArrayPtr = ^VirtualRCArray;
VirtualRCArray =
  object(VirtualArray)
    ...
  end;
```

The VirtualRCArray object, which implements range checking of VirtualArray indices.

Variables

```
EmsAvailable : Boolean;
```

Set to True by OPLARRAY's initialization code if any EMS memory is detected.

Declaration
```
procedure ArrayDimensions(var Rows, Cols : Word);
```

Purpose
Returns the dimensions of the large array.

Description
This routine is especially useful with the LoadA constructor, since the dimensions of the array may not be known in advance.

Example
```
RA.ArrayDimensions(MaxRows, MaxCols);
WriteLn('Array dimensions are ', MaxRows, ' rows by ', MaxCols, ' columns');
```
Displays the actual dimensions of the array RA.

See Also
LoadA

Declaration
```
procedure ClearA(var Value; Initialize : InitType); virtual;
```

Purpose
Initialize all array elements to Value.

Description
Since Value is a var parameter, you will need to assign the initialization value to a variable of the appropriate type and then pass it to ClearA. Initialize can be either ExactInit or FastInit. If FastInit is used, ClearA initializes the entire array to contain the first byte in Value. If ExactInit is used, ClearA assigns the exact Value to each array element.

Example
```
MyElement := 3.14159265;   {approximate value of Pi}
RA.ClearA(MyElement, ExactInit);
```
Initializes each value in RA to Pi.

Done

Declaration

```
destructor Done; virtual;
```

Purpose

Dispose of an array.

Description

All heap space used by the array is disposed.

If the array is an EmsArray or an OpArray acting as an EmsArray, the EMS memory used by the large array is deallocated. Be sure to call Done for all EMS-based arrays before the program halts, since DOS will not automatically deallocate this space.

If the large array is a VirtualArray or an OpArray acting as a VirtualArray, then the array file is updated on disk and the array file closed before disposing of the array. If the VirtualArray had the lDeleteFile option enabled, then the array file is automatically deleted.

Example

```
RA.Done;
```

This example shows a static object calling the Done destructor. Assuming RA is a RamArray, this frees all the heap memory used by RA.

```
Dispose(VAptr, Done);
```

This example shows a dynamic object calling the Done destructor through the Dispose procedure. The heap space for the instance itself, as well as the heap allocated by the object, is deallocated. If VAptr points to a virtual array, the array file is flushed and closed. If the lDeleteFile flag is set, then the array file is deleted.

See Also

Init

Declaration

```
function ErrorA : Word;
```

Purpose

Return last error code and reset internal status variable to zero.

Description

A return value of zero indicates no error. Possible error numbers are:

Error Number	Error Constant	Error Description
0002	ecFileNotFound	file not found
0003	ecPathNotFound	path not found
0008	ecOutOfMemory	insufficient memory
0160	ecDeviceWrite	device write error
0161	ecDeviceRead	device read error
2100	ecInsufficientEms	not enough EMS memory
2101	ecEmsAllocation	error allocating EMS memory
2102	ecNoEms	EMS memory is not installed
2103	ecEmsPageMapping	error mapping EMS page
2104	ecCantFreeEms	error deallocating EMS handle
8050	ecElSizeIsZero	element size is zero
8051	ecIncorrectSize	incorrect dimensions for disk file
8052	ecRowOutOfRange	the specified row is out of range
8053	ecColOutOfRange	specified column is out of range
8054	ecNotLArrayFile	specified file is not a OPLARRAY file
8055	ecElementTooBig	element size is too big
8056	ecBadDimensions	an invalid dimension was specified
8057	ecLessThanOnePage	less than one page fits in RAM
8058	ecFlushError	error attempting to flush array file

This list of error numbers also covers those returned in OPROOT.InitStatus when a constructor fails.

Note that the error number may have an error prefix added, as described in Appendix D.

An alternative to polling ErrorA is to install an error procedure.

Example

```
RA.SetA(Y, X, Value);
Result := RA.ErrorA;
if Result <> 0 then
  WriteLn('Operation failed, Result = , Result);
```

Reports the status of a large array operation. Note that calling ErrorA clears the current error code.

See Also

SetErrorProc

FlushA / GetErrorProc

Declaration

```
procedure FlushA; virtual;
```

Purpose

Flushes any changes to disk (if object type is acting as a VirtualArray).

Description

This method is relevant only to VirtualArrays or OpArrays acting as virtual arrays. It flushes the array's heap buffers and then forces DOS to flush its own file buffers. Turbo Pascal I/O is not used by FlushA, so there is no need to check IoResult.

Example

```
var
  VA : VirtualArray;
...
VA.FlushA;
if VA.ErrorA <> 0 then begin
  WriteLn('Error flushing large array to disk');
  ...
end;
```

Flushes all changes made to VA to disk.

Declaration

```
procedure GetErrorProc(var EP : ErrorProc);
```

Purpose

Get current user error procedure.

Description

This routine allows you to save the existing user error procedure. It is useful if there is a need to save the existing user error procedure before switching to a new one.

Example

```
var
  SaveErrorProc : ErrorProc;
begin
  ...
  RA.GetErrorProc(SaveErrorProc);
  RA.SetErrorProc(MyLErrorProc);
  RA.SetA(101, 2, MyElement);
  RA.SetErrorProc(SaveErrorProc);
  RA.SetA(101, 2, MyElement);
  ...
```

Assuming that this is a 100 row by 100 column array and that RA is a range-checked array object, both of the calls to SetA will fail. The first failure will call MyLErrorProc; the second failure will call the original ErrorProc, whatever that was.

```
var
  SaveErrorProc : Procedure;
...
  RA.GetErrorProc(SaveErrorProc);
```

Incorrect! Define SaveErrorProc as an ErrorProc rather than Procedure. This would generate the compiler error "Type mismatch."

See Also

SetErrorProc

Declarations

```
constructor RamArray.Init(Rows, Cols, ElementSize : Word;
                          HeapToUse : LongInt; ArrayOptions : Byte);
constructor EmsArray.Init(Rows, Cols, ElementSize : Word;
                          HeapToUse : LongInt; ArrayOptions : Byte);
constructor VirtualArray.Init(Rows, Cols, ElementSize : Word;
                              FileName : String;
                              HeapToUse : LongInt; ArrayOptions : Byte);
constructor OpArray.Init(Rows, Cols : Word; ElementSize : Word;
                         FileName : String;
                         HeapToUse : LongInt; ArrayOptions : Byte;
                         var Priority : AutoPriority);
constructor PrimitiveArray.Init(RowsInPage : Word; ColsInPage : Word;
                                PagesDown : Word; PagesAcross : Word;
                                ElementSize : Word; FileName : string;
                                HeapToUse : LongInt; ArrayOptions : Byte);
```

Purpose

Make a new large array.

Description

Rows specifies the maximum number of rows in the array and Cols the maximum number of columns. Since array indices are zero based, the highest valid row and column are one less than the maximum specified. Rows and Cols are used in all constructors but PrimitiveArray.Init.

ElementSize is the size in bytes of the element to be stored in the array. ElementSize is usually passed by using the Turbo Pascal SizeOf operator on your data type. ElementSize can be no larger than 16384 for the EmsArray type, or up to 65521 bytes for any of the other types.

HeapToUse is the maximum number of heap bytes to use for data storage (not including array overhead). If the array's data size is greater than HeapToUse, the constructor will fail. Even if the data size is less than HeapToUse, the constructor will fail if it is unable to allocate all necessary data structures on the heap. Additional structures include a header structure, an array of pointers to pages and, for the virtual array, disk and page indexes for managing the page buffers. For typical cases the extra space is negligible (less than a hundred bytes). For special cases with small page sizes and many pages, the array management overhead may become significant, perhaps thousands or tens of thousands of bytes.

If you want a RamArray to be able to use all available memory, pass MemAvail for HeapToUse. HeapToUse is ignored for EmsArray, since no data is stored on the heap. For a VirtualArray, HeapToUse specifies the maximum amount of heap space to devote to a paging area between disk and your program. As you allot more RAM space, less paging will occur and the array performance will improve. You must allocate enough RAM for at least one page of the array. If the HeapToUse parameter is large enough to hold the entire array in memory, then the disk will be accessed only when the large array file is closed (through a call to the Done destructor), when the FlushA or StoreA method is called, or when the lKeepDiskCurrent option is enabled and SetA is called.

ArrayOptions is used to pass option flags to the Init constructor. ArrayOptions has meaning only for OpArray, VirtualArray, VirtualRCArray, and PrimitiveArray objects. It is passed to all others for consistency. There are two options relevant to VirtualArray, PrimitiveArray and OpArray (acting as a VirtualArray) objects: lKeepDiskCurrent and lDeleteFile. The lRangeCheck option is valid only for OpArray objects. See Constants, above, for more information.

The VirtualArray, OpArray, and PrimitiveArray object types all take a FileName parameter. FileName specifies the name of the file in which to store the large array. This name may optionally include drive and pathname. Disk space must be available for the actual data size of the array plus a small amount of overhead. Init overwrites an existing file of the same name without warning.

With the exception of PrimitiveArray.Init, the constructors automatically determine a page structure for the array. The criteria used to determine the page structure are: each page is limited to 64K bytes, each page forms an identical rectangular subsection of the total array, and the overhead for array management plus wasted space at the edges is to be minimized.

The Priorities parameter of the OpArray object type determines the order in which the constructor attempts to use machine resources to allocate the array. The section under the heading "Array Priorities" above describes the usage of this parameter.

The PrimitiveArray type is provided for circumstances where complete control over the array's page structure is needed. By contrast, the VirtualArray's constructor automatically divides an array into a page structure that it considers optimal in terms of minimizing wasted memory usage. Unfortunately, the automatic constructor doesn't know enough to optimize performance as well. Because the disk device used for paging is so much slower than CPU speed or memory accesses, the number of disk accesses for array paging should be minimized to maintain good array performance. If an array is accessed primarily in row order (that is, the row index is incremented rapidly, and the column number is incremented less often), then each page should hold as many rows as possible. The converse is true if an array is accessed primarily in column order. The choices made automatically by VirtualArray.Init might not be ideal for your program's needs. With the PrimitiveArray.Init constructor, you can assure that a desired organization is used.

Based on the parameters passed to PrimitiveArray.Init, the total number of rows in the array is PagesDown*RowsInPage and the number of columns is PagesAcross*ColsInPage. The total number of pages is PagesDown*PagesAcross, and the size in bytes of each page is RowsInPage*ColsInPage*ElementSize.

The EMS-based arrays allocate almost no heap space; all of the array's data is stored directly in EMS memory. EmsArray.Init calculates the minimum number of 16K pages to hold the data and adds one. (The last page is needed to avoid special case logic when accessing elements near the end of the array.)

Note that none of these constructors initializes array elements. Without further initialization (perhaps by calling ClearA), the array elements will contain random values. Also note that virtual arrays do not allocate disk space for a page until at least one element in the page has been set. You can take advantage of this behavior in some cases by creating an array of maximum capacity and using only the needed portions.

If a constructor fails, InitStatus will contain an explanatory error code.

The constructors for arrays that implement range checking (RamRCArray, EmsRCArray, VirtualRCArray) have identical parameters to the objects from which they are derived.

Examples

```
var RA : RamArray;
...
if not RA.Init(100, 100,          {Rows, columns}
               SizeOf(MyElement), {Size of one element}
               MemAvail,          {Use all available heap}
               0)                 {Options}
then begin
  WriteLn('Failed to init RamArray. Status = ', InitStatus); Halt;
end;
```

Creates a RamArray with 100 rows and 100 columns.

```
var EA : EmsRCArray;
...
if not EA.Init(100, 100,          {Rows, columns}
               SizeOf(MyElement), {Size of one element}
               MemAvail,          {HeapToUse: doesn't matter}
               0)                 {Options}
then begin
    WriteLn('Failed to init EmsRCArray. Status = ', InitStatus); Halt;
end;
```

Creates a range checking EmsArray (EmsRCArray) with 100 rows and 100 columns.

```
var VA : VirtualArray;
...
if not VA.Init(100, 100,           {Rows, columns}
               SizeOf(MyElement),  {Size of one element}
               'TEMP.ARR',         {Virtual array file}
               MemAvail div 2,     {Heap space to use}
               lDeleteFile)        {Array options}
then begin
  WriteLn('Failed to init VirtualArray. Status = ', InitStatus); Halt;
end;
```

Creates a VirtualArray with 100 rows and 100 columns and stores the portion that won't fit on the heap in the disk file TEMP.ARR. The disk file will be deleted automatically when the Done destructor is called.

```
var PA : PrimitiveArray;
...
if not PA.Init(100, 20,            {RowsInPage, ColsInPage}
               1, 20,              {PagesDown, PagesAcross}
               SizeOf(MyElement),  {Size of one element}
               'FLIGHT.DAT',       {Virtual array file}
               1024*12,            {Data paging space to use = 12K}
               0)                  {Array options (none, in this example)}
then begin
  WriteLn('Failed to init PrimitiveArray. Status = ', InitStatus); Halt;
end;
```

Create a virtual array with complete control over the page definitions. We assume that we will usually be processing the array in row order. This page structure puts 20 complete columns in each page, hence minimizing disk paging as we scan through the rows.

```
const MyPri : AutoPriority = (lRamArray, lEmsArray, lVirtualArray);
var   OA    : OpArray;
...
if not OA.Init(100, 100,           {Rows, columns}
               SizeOf(MyElement),  {Size of one element}
               'TEMP.ARR',         {Virtual array file}
               MemAvail div 4,     {Heap space to use}
               0,                  {Array options}
               MyPri)              {Allocation priority}
then begin
  WriteLn('Failed to Init OpArray,  Status = ', InitStatus); Halt;
end;
case OA.TypeOfArray of
  lRamArray : WriteLn('RAM array');
  lEmsArray : WriteLn('EMS array');
  lVirtualArray : WriteLn('Virtual array');
end;
```

Creates an OpArray named OA, then displays the type of array that was actually allocated. MyPri specifies that the array should be placed entirely in RAM if it will fit in MemAvail div 4 bytes, or in EMS if that fails, or paged to disk as a last resort.

See Also

LoadA

LoadA

Declaration

```
constructor RamArray.LoadA(FileName : String;
                           HeapToUse : LongInt; ArrayOptions : Byte);
constructor EmsArray.LoadA(FileName : String;
                           HeapToUse : LongInt; ArrayOptions : Byte);
constructor VirtualArray.LoadA(FileName : String;
                               HeapToUse : LongInt; ArrayOptions : Byte);
constructor OpArray.LoadA(FileName : String;
                          HeapToUse : LongInt; ArrayOptions : Byte;
                          var Priority : AutoPriority);
```

Purpose

Load an existing array file to construct a large array instance.

Description

LoadA first attempts to open the file specified in FileName. If the file does not exist, the LoadA constructor fails. It is the calling program's responsibility to find the file first if necessary. LoadA then reads a header that describes the size of the array. Based on this information, LoadA attempts to allocate the necessary resources (heap or EMS memory) to load the array. Heap usage is limited to the value specified by the parameter HeapToUse (in the same manner as for Init). Then data is transferred from the file into the array object. If the array is being stored in RAM or EMS, the file is closed. If the array is virtual, the file is left open and continues to provide virtual storage.

The resources needed to load the array will vary depending on the array object type used. For example, to load a 100x100 array of words into a RamArray requires 20,000 bytes of heap space (plus a small amount of overhead), whereas to store the same array in EMS would require almost no heap space and three 16K pages of EMS memory.

Large array files stored by any of OPLARRAY's large array objects can be read by the LoadA constructor of any of the other large array types. The LoadA constructor can also read large array files compatible with the TPVARRAY format used by Turbo Professional.

If there are insufficient resources to load the array, then the LoadA constructor will fail. InitStatus will contain an error code indicating the cause of the error.

Examples

```
var RA : RamArray;
...
if not RA.LoadA('RA.ARR', MaxAvail div 2, 0) then begin
  WriteLn('Failed to Load RamArray,  Status = ', InitStatus);
  Halt;
end;
...
```

A large array is loaded from the disk file RA.ARR into a RAM array.

```
const
  MyPri : AutoPriority = (lEmsArray, lRamArray, lVirtualArray);
var OA : OpArray;
...
if not OA.LoadA('GRID.DAT', MaxAvail div 2, 0, MyPri) then
  WriteLn('Failed to Load OpArray,  Status = ', InitStatus);
  Halt;
end;
```

This example attempts to open the large array file GRID.DAT. The priority list MyPri specifies to try EMS first, then the heap, and if all else fails virtual buffering.

See Also

Init

Declaration
```
function lOptionsAreOn(OptionCodes : Word) : Boolean;
```

Purpose
Return True if OptionCodes are enabled.

Description
This routine allows you to determine whether a single option or multiple options are on. In OPLARRAY there are three possible options: lKeepDiskCurrent, lDeleteFile, and lRangeCheck.

Example
```
if VA.lOptionsAreOn(lDeleteFile) then
  {take some action};
```
Takes some action if the lDeleteFile option is on.

Declaration
```
procedure lOptionsOff(OptionCodes : Word);
```

Purpose
Turn specified array options off.

Description
lOptionsOff ignores the lRangeCheck option. Range checking is a property that may be specified only when an OpArray is constructed.

Example
```
VA.lOptionsOff(lDeleteFile);
```
The virtual array's disk file will not be deleted when Done is called.

Declaration
```
procedure lOptionsOn(OptionCodes : Word);
```

Purpose
Turn specified array options on.

Description
lOptionsOn ignores the lRangeCheck option. Range checking is a property that may be specified only when an OpArray is constructed.

Example
```
VA.lOptionsOn(lKeepDiskCurrent+lDeleteFile);
```
The virtual array's disk file will be updated whenever an element is changed, and it will be deleted when Done is called.

RangeError

Declaration

```
function RangeError(Row, Col : Word) : Boolean; virtual;
```

Purpose

Returns True if the specified Row or Col is out of range.

Description

This routine is provided as a means of checking the validity of a row and column before calling SetA or GetA. It works even for non-range-checked types.

Example

```
{...Calculate Row and Column...}
if not RA.RangeError(Row, Col) then
  RA.SetA(Row, Col, MyElement)
else
  {handle range error};
```

Assuming that Row and Col are calculated elsewhere, we call RangeError to make sure they're within the range of this array.

See Also

SetA RetA

Declaration

```
procedure RetA(Row, Col : Word; var Value); virtual;
```

Purpose

Return an array element.

Description

All arrays managed by OPLARRAY are zero-based. By this, we mean that the lowest index on either axis of the array is zero. By the same token, the highest index on either axis is one less than the total number of rows or columns in the array.

RetA returns the current value of the array element at position Row, Col. Since OPLARRAY doesn't know the type of the element (only its size), the value is returned through an untyped var parameter. Your calling routine must pass a variable of the appropriate type or RetA may overwrite memory with disastrous consequences.

If the row or column value is outside of the array bounds and the array is a range-checked type, RetA will call the defined user error procedure and set the last error code (so a call to ErrorA will return the error code).

In some cases it is desirable to have a function (rather than a procedure) to return the values of array elements. The OPLARRAY unit cannot define such a function because at compile time the data type for the array is not defined. Fortunately, it is easy for your application to define one or more such functions to return the specific array types of your application. Here's an example:

```
function RetVal(A : AbstractArray; Row, Col : Word) : MyDataType;
var
  V : MyDataType;
begin
  A.RetA(Row, Col, V);
  RetVal := V;
end;
```

This approach will work whenever MyDataType is a type that functions may return. These types include Byte, ShortInt, Word, LongInt, pointers, Real, 8087 numerics, and strings. Record types may not be returned as function results.

Another approach--a better one in an object-oriented environment--is to derive a new object type that is specific to a particular array configuration. See this section's introduction for an example.

Examples

```
if not RA.Init(100, 100, SizeOf(MyElement), MemAvail, 0) then begin
  WriteLn('Failed to Init RamArray, Status = ',InitStatus);
  Halt;
end;
RA.RetA(0, 0, Value);
```

Reads the first element of the array into Value.

```
MyElement := RetVal(RA, 100, 1);
```

Incorrect! Assuming this is part of the above fragment, we're referencing a non-existent row. In a non-range-checking object (like RamArray) this will return garbage. In a range-checking object (like RamRCArray), an error will be generated.

See Also

SetA

SafeRetA / SafeSetA

Declaration

```
procedure SafeRetA(Row, Col : Word; var Value); virtual;
```

Purpose

Returns an array element, safely ignoring out of range indices.

Description

This method is almost identical to RetA, except that it ignores out of range values for Row or Col. In the event that a bad index is passed, Value is unchanged, and *no error* is generated. This routine will not be used often; it is provided primarily for consistency with the SafeSetA method.

Example

```
function GetVal(Row, Col : Word) : Str20;
  {-Returns one element value as a Str20 value}
var
  S : Str20;
begin
  S := '';
  RA.SafeRetA(Row, Col, S);
  GetVal := S;
end;
...
if not RA.Init(100, 100, SizeOf(Str20), MemAvail, 0) then begin
  WriteLn('Failed to init RamArray. Status = ',InitStatus); Halt;
end;
{...Use the array...}
MyElement := RA.GetVal(100, 1);
```

Attempts to retrieve the value of a non-existent row (100). GetA returns the element S unchanged, which means that GetVal returns an empty string.

See Also

RetA SetA

Declaration

```
procedure SafeSetA(Row, Col : Word; var Value); virtual;
```

Purpose

Sets an array element, safely ignoring out of range indices.

Description

This method is almost identical to SetA, except that it ignores out of range values for Row and Col. In the event that a bad index is passed, no array element is set, and *no error* is generated. This routine is primarily for internal use.

Examples

```
if not RA.Init(100, 100, SizeOf(MyElement), MemAvail, 0) then begin
  WriteLn('Failed to init RamArray. Status = ',InitStatus); Halt;
end;
RA.SafeSetA(1, 1, MyElement);
```

Moves the value in MyElement to row one, column one of RA.

```
RA.SafeSetA(100, 1, MyElement);
```

Attempts to set a non-existent element of the array. (This is probably a programming error.) Since we're using SafeSetA, this will return without overwriting memory or setting an error code.

See Also

RetA SetA

Declaration

```
procedure SetA(Row, Col : Word; var Value); virtual;
```

Purpose

Set an array element at specified Row and Col to Value.

Description

All arrays managed by OPLARRAY are zero-based. By this, we mean that the lowest index on either axis of the array is zero. By the same token, the highest index on either axis is one less than the total number of rows or columns in the array.

SetA sets the value of the array element at position Row, Col. Since OPLARRAY doesn't know the type of the element (only its size), the value is passed as an untyped var parameter. Your calling routine must pass a variable of the appropriate type.

If the row or column value is outside of the array bounds and a range-checked array is being used, SetA will call the defined user error procedure and set the last error code (so a call to ErrorA will return the error code).

In some cases it may be desirable to pass new array elements by using constants or expressions. OPLARRAY cannot support this directly because an array's data type is not known at compile time, but you can easily define one or more procedures to set the specific data types of your application. Here's an example:

```
procedure SetVal(A : AbstractArray; Row, Col : Word; V : MyDataType);
begin
  A.SetA(Row, Col, V);
end;
```

SetVal can then be called with constants or expressions in the V parameter.

Another approach--a better one in an object-oriented environment--is to derive a new object type that is specific to a particular array configuration. See this section's introduction for an example.

Examples

```
if not RA.Init(100, 100, SizeOf(MyElement), MemAvail, 0) then begin
  WriteLn('Failed to Init RamArray, Status = ',InitStatus);
  Halt;
end;
MyElement := 'Object';
RA.SetA(1, 1, MyElement);
```

Stores the value 'Object' in row 1, column 1 of RA.

```
RA.SetA(100, 1, MyElement);
```

Incorrect! This example tries to store an element in a non-existent row. In a non-range-checking object (like RamArray), this will corrupt memory. With a range checking object (like RamRCArray), this will fail with a RowOutOfRange error.

See Also

RetA SafeSetA

SetErrorProc

Declaration

```
procedure SetErrorProc(EP : ErrorProc);
```

Purpose

Sets up a user error procedure.

Description

OPLARRAY calls an error procedure whenever an error occurs. The default error procedure does nothing unless the DefBeepOnError constant is True, in which case it beeps. Use SetErrorProc to have OPLARRAY call a user-written routine instead.

The specified error procedure must be compiled with the FAR model (using {$F+}), must be global, and must have parameters that match the following prototype:

```
{$F+}
procedure MyErrorProc(UnitCode : Byte; var ErrorCode : Word;
                      ErrorMsg : string);
```

This procedure type is defined in OPROOT and is consistent with all other **Object Professional** error routines. When OPLARRAY calls it, it passes the value ucLArray for UnitCode. ErrorCode will be a value consistent with the codes described under the ErrorA method and also in Appendix D. ErrorMsg is always the empty string for OPLARRAY errors.

See DisplayLErrorProc in Procedures and Functions below for a standard error handler that displays text error messages.

Example

```
{$F+}
procedure MyErrorHandler(UnitCode : Byte; var ErrorCode : Word;
                         ErrorMsg : String);
const
  ClassOfError : Array[0..4] of String[9] = ('fatal','non-fatal','warning',
                                             'message','no error');
var Seriousness, Err : Word;
begin
  Seriousness := ErrorCode div 10000;  {determine error class}
  Err := ErrorCode mod 10000;          {strip error class from code}
  WriteLn('Whoops, error ', Err, ' has occurred!');
  WriteLn('This is of error class: ', ClassOfError[Seriousness]);
  if UnitCode = ucLArray then
    DisplayLErrorProc(UnitCode, ErrorCode, ErrorMsg)
  else
    WriteLn(ErrorMsg);
  if Seriousness = etFatal then
    Halt;
end;
{$F-}
...
RA.SetErrorProc(MyErrorHandler);
```

Installs MyErrorHandler as the user error procedure for RA. MyErrorHandler first determines the seriousness of the error, extracts the raw error number from the ErrorCode parameter, and displays these numbers. If MyErrorHandler is called by the OPLARRAY unit, then it calls DisplayLErrorProc to display a message indicating the cause of the error. Finally, if it determines that the error class is etFatal, the program is aborted.

See Also

DisplayLErrorProc
ErrorA
GetErrorProc

Declaration
```
procedure StoreA(FileName : String); virtual;
```

Purpose
Write a large array to disk.

Description
StoreA creates a new file as specified by the parameter FileName. StoreA overwrites an existing file of the same name without warning.

There is little point in calling StoreA for a virtual array object. The format used by StoreA is the same as the format used for the virtual array, so StoreA simply flushes the current array and then does a byte-level copy to the specified FileName.

StoreA first writes a header record, describing the array's dimensions and type. Then StoreA writes each of the array's pages, and finally closes the file. StoreA does not deallocate the array's image on the heap. If storing an array is the last need for it in a particular program, call the Done destructor to deallocate the RAM space after the array is written.

The disk file size will be somewhat larger than the actual array size, which is given by

```
ArraySize = TotalRows*TotalCols*ElementSize.
```

StoreA will return an error if it is unable to create, write to, or close the array file.

Files written with StoreA may be read by the LoadA constructor of any of the large array object types.

Example
```
RA.Init(100, 100, SizeOf(MyElement), MemAvail, 0);
{...}
RA.StoreA('RA.ARY');
Result := RA.ErrorA;
if Result <> 0 then
  WriteLn('Failed to store array, Result = ', Result);
RA.Done;
```

Writes the array to disk.

See Also
LoadA

Declaration
```
function TypeOfArray : ArrayType;
```

Purpose
Returns the type of array.

Description
This method is useful when applied to an OpArray object, since this object dynamically determines what type of array to become at runtime.

The lNoArray value should never be returned.

Example
```
case LargeArray.TypeOfArray of
  lRamArray : WriteLn('RAM array');
  lEmsArray : WriteLn('EMS array');
  lVirtualArray : WriteLn('Virtual array');
end;
```

Reports LargeArray's array type.

DisplayLErrorProc

Declaration

```
procedure DisplayLErrorProc(UnitCode : Byte; var ErrorCode : Word;
                            ErrorMsg : String);
```

Purpose

An error procedure that displays a standard message for each large array error.

Description

This routine is provided as an alternative to the simplistic default error handler. It displays an English error message (using WriteLn) that indicates the cause of the error. If the typed constant DefBeepOnError is True, it also issues a short beep. This routine does not halt the program.

Example

```
RA.SetErrorProc(DisplayLErrorProc);
```

Tells RA to use the built-in error procedure DisplayLErrorProc.

See Also

GetErrorProc SetErrorProc

9. String Handling

A. Overview

Almost every commercial library contains a variety of routines for string manipulation, and **Object Professional** is no exception. As expected, you'll find the usual routines for converting numbers to strings in a variety of formats, changing the case of alphabetic characters (including those in the international character set) in a string, trimming leading and trailing blanks, padding strings with blanks, and doing comparisons of strings. But you'll also find a significant number of useful routines that you probably didn't expect.

OPSTRING consists primarily of general purpose string manipulation routines, including all the categories mentioned above. It also contains a number of slightly more unusual routines. Most notable is a complete set of routines for massaging DOS file and path names. These allow you to add and remove extensions and path identifiers, clean up filenames with extraneous characters, and turn valid file names into complete pathnames in Drive:\Path\Filename.Ext format. OPSTRING also has routines to allocate and deallocate heap space for string variables; to search large arrays for the location of a specified string; and to convert strings to Soundex codes, allowing you to search for names that almost match, such as "Smith" and "Smythe".

One of the most flexible routines in the unit is Form, modelled on the Form function in the BCD version of Turbo Pascal 3.0. Form takes two parameters, a string Mask and a real, and returns a formatted string that merges individual digits from the real into the Mask. For example, given the Mask '$##,###,###.##' and the real value 12300.26666, Form would return ' $12,300.27'. (Note that OPBCD offers a comparable function for BCD reals.)

One fairly common complaint about Turbo Pascal is that its strings are limited to 255 characters. Although 255 is usually more than adequate, when you need longer strings, you're stuck. OPASCIIZ offers a comprehensive solution to this problem. Unlike normal strings, an ASCIIZ string has no length byte. Instead, it marks the end of a string with a single null character (#0). The big disadvantage of ASCIIZ strings is that certain operations, such as checking the length of a string, take more time, but the great advantage is that they can be much, much longer. (1024 characters by default, but up to 65,520 if desired.) Of course, the other disadvantage of using ASCIIZ strings is that the Turbo compiler does not support them, but OPASCIIZ contains just about everything you might need to work with them. Besides providing equivalents for Turbo's own string manipulation routines--Concat, Insert, Delete, Pos, Length--it also contains equivalents for most of the basic routines in OPSTRING. Equally important, it contains routines for reading and writing ASCIIZ strings to and from text files.

Tiny OPSTRDEV serves a single function: it lets you use the power of the Write statement to create formatted strings. It is particularly useful if you are using FastWrite (OPCRT), which won't take a number as a parameter, and won't pad strings with blanks. For example, if you wanted to convert the values -357 and 41 into a single string 18 characters wide, you could use the Long2Str and Pad functions in OPSTRING, then send the cumulative result of the several function calls to FastWrite. If you were using OPSTRDEV, however, you could simply say

```
Write(TpStr, -357, 41:14);
FastWrite(ReturnStr, Row, Column, Attribute);
```

More generally, OPSTRDEV allows you to use the formatting capabilities provided by Write statements in situations where you are not writing to the screen or to a text file.

B. OPSTRING: String Manipulation

As mentioned previously, OPSTRING contains all of the string manipulation routines that you would expect, and a number that you might not expect. We think it has just about everything a professional program will need for basic string manipulation. OPSTRING works with Turbo's standard 255 character strings. If you need something longer, check out the OPASCIIZ chapter.

The routines in OPSTRING fall into several categories: numeric conversion and formatting; general purpose manipulation such as case changes, padding, trimming and centering; string comparison and searching; dynamic storage of strings on the heap; and DOS pathname parsing.

Types

```
BTable = array[0..255] of Byte;
```

A table used by the Boyer-Moore search routines.

```
CharSet = set of Char;
```

A set of characters.

```
CompareType = (Less, Equal, Greater);
```

The result of a comparison of two variables.

```
ExtStr = string[ExtLen];
```

The extension portion of a DOS filename.

```
{$IFOPT N+}
Float = Extended;
{$ELSE}
Float = Real;
{$ENDIF}
```

If OPSTRING is compiled with {$N+}, Float is defined as an Extended rather than a Real. The definition of Float affects the Real2Str and Str2Real routines.

```
StringPtr = ^String;
```

A pointer to a string variable.

Constants

```
CommaForPeriod : Boolean = False;
```

If True, the Form function will return '.' for ',' and vice versa. This makes it a bit easier to write programs that can be used both in the U.S. and in countries that use commas for decimal points.

```
ExtLen = 3;
```

Length of the extension portion of a DOS filename.

```
LetterValues : array['A'..'Z'] of Byte =
(3 {A} , 6 {B} , 5 {C} , 4 {D} , 3 {E} , 5 {F} , 5 {G} , 4 {H} , 3 {I} ,
 8 {J} , 7 {K} , 4 {L} , 5 {M} , 3 {N} , 3 {O} , 5 {P} , 7 {Q} , 4 {R} ,
 3 {S} , 3 {T} , 4 {U} , 6 {V} , 5 {W} , 8 {X} , 8 {Y} , 9 {Z} );
```

A table used by CompareLetterSets for estimating word similarity.

```
MoneySign : Char = '$';
```

This character is used when displaying values in dollars-and-cents format. If desired, you could change this constant to #156, for example, allowing Form to display amounts in terms of British pounds rather than dollars.

Declaration

```
function AddBackSlash(DirName : string) : string;
```

Purpose

Add a default backslash to a directory name.

Description

A backslash character '\' separates an DOS directory name from a filename contained within that directory. A special case prevents the simple concatenation of a backslash to all directory names, however. The root directory name actually consists of a backslash, and DOS becomes unhappy if an additional backslash is appended. AddBackSlash concatenates a backslash to a directory name only when the directory name does not already end in a backslash.

Examples

```
PathName := AddBackSlash('C:\')+'AUTOEXEC.BAT';
```

The resulting PathName is 'C:\AUTOEXEC.BAT'

```
PathName := AddBackSlash('\turbo\test')+'BUG.PAS';
```

The resulting PathName is '\turbo\test\BUG.PAS'

See Also

FullPathName

Declaration

```
function BinaryB(B : Byte) : string;
```

Purpose

Return binary string for Byte.

Description

Returns an 8 character string which represents the binary value of the byte. Leading zeros are included.

Examples

```
BinaryB(0)
```

Returns '00000000'

```
BinaryB($B1)
```

Returns '10110001'

Declaration

```
function BinaryL(L : LongInt) : string;
```

Purpose

Return binary string for LongInt.

Description

Returns a 32 character string which represents the binary value of a LongInt. No separation characters are inserted within the string. Leading zeros are included.

Example

```
BinaryL(-1)
```

Returns '11111111111111111111111111111111'

```
BinaryL(7)
```

Returns '00000000000000000000000000000111'

BinaryW / BMMakeTable

Declaration

```
function BinaryW(W : Word) : string;
```

Purpose

Return binary string for Word.

Description

Returns a 16 character string which represents the binary value of a Word. No separation characters are inserted within the string. Leading zeros are included.

Examples

```
BinaryW($B800)
```

Returns '1011100000000000'

```
BinaryW(-1)
```

Incorrect! This won't compile because Words are always non-negative. Use BinaryL if negative numbers are needed.

See Also

BinaryB BinaryL

Declaration

```
procedure BMMakeTable(MatchString : string; var BT : BTable);
```

Purpose

Build Boyer-Moore link table.

Description

The Boyer-Moore algorithm is a very fast string search method. It gains this speed from the fact that it need not compare many elements of the string. The elements it can skip are determined by a link table that must be initialized before the search itself begins.

The link table is an array[0..255] of byte. For each character in the ASCII set (0..255) the table contains the number of positions the search can safely skip. BMMakeTable takes the string to be matched, MatchString, and returns a properly initialized link table. The link table will be passed to the Boyer-Moore search routine itself to aid in the search.

The BMMakeTable operation may dominate the performance of B-M if the region to be searched is short. In general, you should minimize the number of calls to BMMakeTable. If you find the need to call it very often, you will probably get better performance using the brute force function Search.

See the BMSearch function for an example.

See Also

BMSearch BMSearchUC

Declaration

```
function BMSearch(var Buffer; BufLength : Word;
                  BT : BTable; MatchString : string) : Word;
```

Purpose

Search Buffer for MatchString using the Boyer-Moore algorithm.

Description

This string search routine returns results consistent with the Search function, also found in OPSTRING. However, BMSearch uses the Boyer-Moore algorithm, which in some cases leads to significantly better performance than Search.

BMSearch differs from Search in the parameters that it requires. Like Search, it will search any Buffer region, with a BufLength of up to 65534 bytes. Unlike Search, it requires a special table (BT) previously initialized by BMMakeTable, and the search string is a normal Turbo string.

BMSearch returns the number of bytes searched to find the target string. If the string is not found, BMSearch returns $FFFF. If the target string is found at the very start of Buffer, BMSearch returns 0.

The performance of Boyer-Moore searching is dependent on many factors. If the buffer region is the size of a normal Turbo string (255 bytes or less), and the link table BT must be regenerated for each search, BMSearch will always be slower than the simpler Search routine. On the other hand, if the buffer is large, or if the same link table can be used many times, BMSearch's performance ranges from about the same as Search's to twice as fast, depending on the length of the MatchString. Typically, the longer the MatchString, the more advantage BMSearch will gain.

Special thanks to Don Strenczewilk for providing the initial version of the Boyer-Moore search routines.

Example

```
function BMS(var Buffer; BufLength : Word; MatchSt : string) : Word;
var
  LocalTable : BTable;
begin
  BMMakeTable(MatchSt, LocalTable);
  BMS := BMSearch(Buffer, BufLength, LocalTable, MatchSt);
end;
```

This function hides the link table from the calling program. Unfortunately, it also updates the link table at every call, thus minimizing the performance of the B-M search. This function will provide good performance only if the Buffer region is large. Otherwise, an application should call BMMakeTable explicitly and use the returned link table as many times as possible before discarding it.

See Also

BMMakeTable
BMSearchUC
Search

BMSearchUC / Center

Declaration

```
function BMSearchUC(var Buffer; BufLength : Word;
                    BT : BTable; MatchString : string) : Word;
```

Purpose

Case-insensitive search of Buffer for MatchString using the Boyer-Moore algorithm.

Description

This function operates just like BMSearch, except that searching is not case sensitive. See BMSearch for more details.

Important requirement: It is the application's responsibility to assure that MatchString is raised to upper case prior to calling either BMMakeTable or BMSearchUC.

BMSearchUC almost always outperforms SearchUC, even if the link table BT must be built for each search. The speed of BMSearchUC ranges from slightly slower than SearchUC to about four times faster depending on how often the link table is rebuilt, the length of the MatchString, and the data contents.

Note that BMSearchUC handles correctly all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

Example

```
function UCBMS(var Buffer; BufLength : Word; MatchSt : string) : Word;
var
  LocalTable : BTable;
begin
  MatchSt := StUpCase(MatchSt);      {Guarantee that MatchSt is uppercase}
  BMMakeTable(MatchSt, LocalTable); {Build the link table}
  UCBMS := BMSearchUC(Buffer, BufLength, LocalTable, MatchSt);
end;
```

This function hides from the calling program all operations preparatory to a case-insensitive Boyer-Moore search. If the link table could be used more than once, this function will waste performance, although it will still be faster than most alternative approaches.

See Also

BMMakeTable
BMSearch
SearchUC

Declaration

```
function Center(S : string; Width : Byte) : string;
```

Purpose

Return a string centered in a blank string of specified width.

Description

If the length of S is greater than Width, then S is returned unchanged. If S is an empty string, a string of blanks is returned.

Example

```
Center('hello', 10)
```

Returns the string ' hello '

See Also

CenterCh

Declaration
```
function CenterCh(S : string; Ch : Char; Width : Byte) : string;
```

Purpose
Return a string centered in a string of Ch with specified width.

Description
If the length of S is greater than Width, then S is returned unchanged. If S is an empty string, a string of characters Ch is returned.

Example
```
CenterCh('hello', '-', 10)
```
Returns the string '--hello---'

Declaration
```
function CharStr(Ch : Char; Len : Byte) : string;
```

Purpose
Return a string of length Len filled with Ch.

Description
If Len is zero, an empty string is returned.

Example
```
CharStr('-',10)
```
Returns '----------'

See Also
PadCh

Declaration
```
function CleanPathName(PathName : string) : string;
```

Purpose
Return a pathname cleaned up as DOS will do it.

Description
You may have noticed that DOS will accept filenames containing more than 8 characters in the name, or more than 3 in the extension. It simply truncates any excess characters. The same behavior applies to subdirectory names, whether given alone or as part of a pathname.

If your application displays filenames, it will look silly to display characters ignored by DOS. CleanPathName automatically truncates any extra characters.

Examples
```
CleanPathName('AUTOEXEC.BAT')
```
Returns 'AUTOEXEC.BAT'
```
CleanPathName('AUTOEXECA.BAT1')
```
Returns 'AUTOEXEC.BAT'
```
CleanPathName('D:\TEST12345\BUG.PAS')
```
Returns 'D:\TEST1234\BUG.PAS'

See Also
FullPathName

CompareLetterSets

Declaration

```
function CompareLetterSets(Set1, Set2 : LongInt) : Word;
```

Purpose

Returns the sum of the values of the letters common to Set1 and Set2.

Description

When used in conjunction with MakeLetterSet, CompareLetterSets provides an alternative to the Soundex algorithm for judging the similarity of two words or strings of words. The algorithm was developed by Michael Bickel, and it is discussed at length in Jim Howell's "An Alternative to Soundex" (*Dr. Dobb's Journal*, November 1987).

Before calling CompareLetterSets, you must use MakeLetterSet to create a bit-mapped long integer (or "letter set") indicating what letters are used in a particular string; this process should be repeated for each string to be compared. Given two such sets, CompareLetterSets will then find the union of the two sets (the letters shared in common) and return a value that indicates how closely the two strings match. In calculating this value, it relies on the LetterValues table (declared in OPSTRING.PAS), which assigns relatively high values to rarely used letters ('J', 'Q', 'X', 'Z') and low ones to the most frequently used letters ('A', 'E', 'N', 'S', 'T'). The return value is simply the sum of the values for each set in the union. To determine the maximum possible value that could be returned in any given situation, you would compare the letter set for a string to itself:

```
PossibleValue := CompareLetterSets(LetterSet1, LetterSet1);
```

Comparisons to the letter sets for other strings will yield a result in the range 0..PossibleValue. The higher the value, the higher the degree of similarity.

Unfortunately, there is no definitive method for deciding whether or not the value returned by CompareLetterSets indicates a close similarity, so you'll probably need to run some experiments to determine how best to use these two functions. One way to make its results more informative would be to use a function such as DissimilarityIndex, shown in the example below, which returns a value between 0.0 (high degree of similarity) and 130.0 (first string is empty). (Similar strings will usually have an index of 0.3 or less.) You might also wish to modify the function to take into account factors that Bickel's algorithm ignores (are the lengths of the two strings the same? are the first characters the same? etc.). For a more complete discussion of the algorithm, see Howell's article in Dr. Dobb's.

Example

```
function DissimilarityIndex(S1, S2 : string) : real;
var Set1, Set2 : LongInt;
    Possible, Actual, Delta : Integer;
begin
  Set1 := MakeLetterSet(S1);
  Set2 := MakeLetterSet(S2);
  Possible := CompareLetterSets(Set1, Set1);
  Actual := CompareLetterSets(Set1, Set2);
  Delta := Possible - Actual;
  if Possible = 0 then
    DissimilarityIndex := 130
  else
    DissimilarityIndex := Abs(Delta/Possible);
end;
var S1, S2 : string;
begin
  repeat
    Write('String 1: '); ReadLn(S1);
    Write('String 2: '); ReadLn(S2);
    WriteLn('Index:  ', DissimilarityIndex(S1, S2):1:3);
  until False; {press ^C to stop}
end.
```

Declaration

```
function CompString(S1, S2 : string) : CompareType;
```

Purpose

Return less, equal, greater if S1 < S2, S1 = S2, or S1 > S2.

Description

CompString does the same thing as Turbo Pascal's string comparison infix operators, with one advantage. If an application branches to any of three statements depending on whether the comparison shows "equal", "greater", or "less", CompString will make the determination in a single comparison, whereas the infix operators will require at least two comparisons. Thus, for certain programs, CompString will offer a speed advantage.

Examples

```
var
  l1, l2 : string;
case CompString(l1, l2) of
  Less :   writeln('l1 < l2');
  Equal:   writeln('l1 = l2');
  Greater: writeln('l1 > l2');
end;
```

This general structure may be used when three different outcomes hinge on a single string comparison.

```
CompString('aaa', 'aaaa')
```

Returns Less.

```
CompString('aaab', 'aaaa')
```

Returns Greater.

```
CompString('Aaaa', 'aaaa')
```

Returns Less.

See Also

CompStruct CompUCString

CompStruct / CompUCString

Declaration

```
function CompStruct(var S1, S2; Size : Word) : CompareType;
```

Purpose

Compare two fixed-size structures.

Description

Pascal does not provide built-in support for comparison of structured data types, such as arrays or records. In some cases this is a sensible restriction. For example, consider a record containing an integer followed by a string. A comparison routine would have no reasonable way of knowing that a string was embedded, and as a result any garbage characters beyond the current length of the string would play a role in the comparison.

In spite of these fundamental difficulties, comparison of structures is often desirable. The CompStruct function offers a solution. Although CompStruct returns a result from the enumerated type Less, Equal, Greater, in most cases only equality or non-equality will be significant. CompStruct uses ASCII comparison on a byte-by-byte basis to determine Less or Greater status.

Note that the problem regarding embedded strings applies to CompStruct. It is the programmer's responsibility to avoid this situation.

The Size parameter to CompStruct is the number of bytes to compare. The best way to determine Size is with the built-in Turbo operator SizeOf().

Examples

```
type
  ArrayType = array[1..3] of byte;
const
  A1 : ArrayType = (0, 1, 2);
  A2 : ArrayType = (0, 1, 3);
  A3 : ArrayType = (0, 1, 2);
CompStruct(A1, A2, SizeOf(ArrayType))
```

Returns Less

```
CompStruct(A1, A3, SizeOf(ArrayType))
```

Returns Equal

Declaration

```
function CompUCString(S1, S2 : string) : CompareType;
```

Purpose

Compare two strings in a case-insensitive manner.

Description

CompUCString converts lowercase characters to upper case prior to comparing two strings. Although slower than CompString or Turbo's built-in string comparison, CompUCString is much faster than calling StUpCase and then performing a normal string comparison. Note that CompUCString handles correctly all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

Examples

```
CompUCString('AAA', 'aaaa')
```

Returns Less.

```
CompUCString('AAAB', 'aaaa')
```

Returns Greater.

```
CompUCString('Aaaa', 'aaaa')
```

Returns Equal.

Declaration
```
function DefaultExtension(Name : string; Ext : ExtStr) : string;
```

Purpose
Return a filename or pathname with a default extension attached.

Description
If Name already has an extension, no change is made. Otherwise, Ext is appended to it. Ext should be a maximum of three characters and not begin with a period.

Examples
```
DefaultExtension('TEST', 'pas')
```
Returns 'TEST.pas'
```
DefaultExtension('TEST.', 'PAS')
```
Returns 'TEST.'

See Also
ForceExtension

Declaration
```
function Detab(S : string; TabSize : Byte) : string;
```

Purpose
Expand tabs in string S to blanks on spacing TabSize.

Description
DeTab removes tabs from a string, converting them into appropriate strings of blanks. Unlike EnTab, DeTab will apply its conversion to all parts of a string, whether or not the string contains quoted literals. A TabSize of 8 is the normal size used by most text editors. If TabSize is zero, DeTab will always return an empty string.

Example
```
var
  S : string;
begin
  while not EOF(Input) do begin
    ReadLn(Input, S);
    WriteLn(Output, DeTab(S, 8));
  end;
end.
```
Implements a complete filter for expanding tabs in a text file.

See Also
EnTab

DisposeString / Entab

Declaration

```
procedure DisposeString(P : StringPtr);
```

Purpose

Deallocate space for string at p.

Description

Applications that perform extensive text processing are often better off storing strings on the heap rather than as global variables. The heap offers two advantages in this situation: it can grow to use all 640K of DOS memory, and strings can use just the memory they need rather than the maximum possible length. For example, suppose a filter program must read in each line of a text file and store it in memory prior to processing. By storing each line as global data, even one character would consume the maximum line length. By storing the lines on the heap, each string would consume exactly what it needed. DisposeString works in concert with StringToHeap. StringToHeap allocates heap space for a specified string and returns a pointer to the string. DisposeString returns the heap space associated with that pointer to the free memory pool. If the pointer passed to DisposeString is nil, no action is performed.

Example

```
var
  P : pointer;
  S : string;
begin
  P := StringToHeap('hello');
  S := StringFromHeap(P);
  DisposeString(P);
end;
```

The first StringToHeap statement allocates 6 bytes of heap space for 'hello' plus its length byte. StringFromHeap loads 'hello' into S. DisposeString releases the 6 bytes of heap space.

See Also

StringFromHeap StringToHeap

Declaration

```
function Entab(S : string; TabSize : Byte) : string;
```

Purpose

Convert blanks in a string to tabs on spacing TabSize.

Description

EnTab compresses a string by converting runs of spaces to appropriate sequences of tabs and spaces. EnTab recognizes literal strings delimited by single quotes or double quotes, and does not convert spaces to tabs within these strings. It also recognizes the C "escape" character '\' to allow quotes embedded within strings. A TabSize of 8 is the normal size used by most text editors. If TabSize is zero, Entab will always return an empty string.

Example

```
var S : string;
begin
  while not EOF(Input) do begin
    ReadLn(Input, S);
    WriteLn(Output, EnTab(S, 8));
  end;
end.
```

Implements a complete filter for compressing text files by tab conversion.

See Also

DeTab

Declaration

```
function ExtractWord(N : Byte; S : string; WordDelims : CharSet) : string;
```

Purpose

Given a set of word delimiters, return the N'th word in S.

Description

This routine returns the N'th word in the specified string. WordDelims is a user-defined set of word delimiters used to determine where words begin and end. At the very least, this set should contain ' '. If S contains fewer than N words, the string returned will be empty.

Example

```
S := 'Four score and seven years ago, our forefathers...';
SixthWord := ExtractWord(6, S, [' ', ',', '.']);
```

The result would be 'ago', since ',' is a word delimiter.

See Also

WordCount WordPosition

Declaration

```
function ForceExtension(Name : string; Ext : ExtStr) : string;
```

Purpose

Force the specified extension onto the file name.

Description

Whether or not the given name has an extension, ForceExtension returns a new filename with the specified extension.

Examples

```
ForceExtension('TEST', 'pas')
```

Returns 'TEST.pas'

```
ForceExtension('TEST.', 'PAS')
```

Returns 'TEST.PAS'

```
ForceExtension('C:\AUTOEXEC.BAT', 'PAS')
```

Returns 'C:\AUTOEXEC.PAS'

See Also

DefaultExtension

Form

Declaration

```
function Form(Mask : string; R : Float) : string;
```

Purpose

Returns a formatted string with digits from R merged into the Mask.

Description

Form provides the equivalent of Turbo 3 BCD's Form function for 6-byte or 8087 reals in Turbo Pascal version 5.5 or higher. The Form function offers advanced numeric formatting especially applicable to financial reports.

Since Pascal does not allow user-defined routines to accept a variable number of parameters, the **Object Professional** Form function accepts only a single parameter of type Real.

The Mask parameter is a string that may contain any combination of normal characters as well as a sequence of the special characters # * @ $ - + , . This sequence is called a numeric field, and Form will replace it with a formatted version of the real number. The number is returned right-justified within the field, decimals are returned rounded if they exceed the number of decimals specified by the format, and if the number is too large to be returned in the field, all digit positions are filled with asterisks.

The special characters are interpreted as follows:

\# A digit position. If the numeric field contains no * or @ characters, unused digits are returned as blanks. If the numeric field contains no sign positions (+ or - characters) and the number is negative, a floating minus is returned in front of the number. Examples:

```
Form('####', 34.567) returns '  35'
Form('####.##', -12.3) returns ' -12.30'
Form('##.##', 1000.0) returns '**.**'
```

* A digit position. Unused positions are returned as asterisks instead of blanks. The * character needs occur only once in the numeric field to activate this effect. The sign of the number will not be returned unless the field also contains a sign character (- or +). Examples:

```
Form('*##.#', 4.567) returns '**4.57'
Form('****', 123) returns '*123'
```

@A digit position. Unused positions are returned as zeros instead of blanks. The @ character needs occur only once in the numeric field to activate this effect. The sign of the number will not be returned unless the field also contains a sign character (- or +). Examples:

```
Form('@##', 9) returns '009'
Form('@@@.@@', 12.3) returns '012.30'
```

$ A digit position. A floating dollar sign is returned in front of the number. The $ character needs occur only once in the numeric field to activate this effect. Examples:

```
Form('$####.##', 123.45) returns ' $123.45'
Form('*$####.##', 12.34) returns '***$12.34'
```

\- A sign position. If the number is negative, a minus will be returned in that position. If it is positive, a blank will be returned. Examples:

```
Form('-###.##', -1.2) returns '-  1.20'
Form('-###.##', 1.2) returns '   1.20'
```

\+ A sign position. If the number is positive, a plus will be returned in that position. If it is negative, a minus will be returned. Examples:

```
Form('+###.##', 1.2) returns '+  1.20'
Form('*$####.##+', -12.34) returns '***$12.34-'
```

, A decimal comma or a separator comma.

. A decimal period or a separator period. The last period or comma in the numeric field is considered the decimal delimiter. Examples:

```
Form('##,###,###.##', 12345678.91) returns '12,345,678.91'
Form($#.###.###,##', -12345.6) returns '  -$12.345,60'
```

Note: The most frequently asked question about Form concerns cases like the following:

```
WriteLn( Form('###,###,###', 1000.0) );
```

What many users expect is ' 1,000'. What they get is ' 1,000,000'. This is *not* a bug. It is consistent with the behavior of the Form function in TURBOBCD 3.0, which treated the last period *or comma* in the mask as marking the decimal point, allowing Form to be used in countries that use commas in places where Americans use periods. To get the expected result, you would need to add a final decimal point to the mask ('###,###,###.') and then remove it from the string returned by Form (this is, in effect, what LongIntForm does).

See Also

Form (OPBCD) LongIntForm

FullPathName / HexB

Declaration

```
function FullPathName(FName : string) : string;
```

Purpose

Given FName (known to exist), return a full pathname.

Description

FullPathName does not search the DOS PATH to find a file (see ExistOnPath for that operation). Instead, FullPathName returns a complete DOS pathname from a partial one that already allows DOS to find the file. The pathname will include a drive letter, complete directory, and filename.

In many cases, FullPathName must interrogate DOS to determine the current directory on the current or specified drive. For this purpose, a disk must have been loaded into the drive or a DOS critical error will occur.

Examples

```
FullPathName('AUTOEXEC.BAT')
```

Returns 'C:\AUTOEXEC.BAT' if the current drive is C:

```
FullPathName('D:..\TEST\CHECK.PAS')
```

Returns 'D:\EXAMPLES\..\TEST\CHECK.PAS' if the current directory on drive D: is \EXAMPLES

See Also

ExistOnPath (OPDOS)

Declaration

```
function HexB(B : Byte) : string;
```

Purpose

Return hex string for Byte.

Description

Returns a 2-character hexadecimal string for the byte. Leading zeros are included. No '$' or 'h' characters are included.

Examples

```
HexB(0)
```

Returns '00'

```
HexB(250)
```

Returns 'FA'

See Also

HexL HexW

Declaration
```
function HexL(L : LongInt) : string;
```

Purpose
Return hex string for LongInt.

Description
Returns an 8-character hexadecimal string for the LongInt. Leading zeros are included. No '$' or 'h' characters are included.

Examples
```
HexL(0)
```
Returns '00000000'
```
HexL(655360)
```
Returns '000A0000'

Declaration
```
function HexPtr(P : Pointer) : string;
```

Purpose
Return hex string for pointer.

Description
Given a pointer, HexPtr returns a string of the form 'ssss:oooo', where 'ssss' is the segment portion of the address, and 'oooo' is the offset.

Example
```
if (ErrorAddr <> nil) or (ExitCode <> 0) then begin
  {don't report ^Break as an error}
  if ExitCode <> $FF then
    WriteLn('Error ', HexW(ExitCode), ' at ', HexPtr(ErrorAddr));
  {prevent runtime library from reporting error again}
  ErrorAddr := nil;
  ExitCode := 0;
end;
```
Code such as the above might be included in a user-defined exit handler to report runtime errors. HexPtr is used to turn the address of the error into a string.

Declaration
```
function HexW(W : Word) : string;
```

Purpose
Return hex string for Word.

Description
Returns a 4-character hexadecimal string for the word. Leading zeros are included. No '$' or 'h' characters are included.

Examples
```
HexW(0)
```
Returns '0000'
```
HexW(MaxInt)
```
Returns '7FFF'

JustExtension / JustFilename

Declaration

```
function JustExtension(Name : string) : ExtStr;
```

Purpose

Return just the extension of a pathname.

Description

Drive, directory and filename are stripped from a pathname, returning just the extension. The extension does not include a period. If the input string has no extension, the empty string is returned.

Examples

```
JustExtension('TEST.PAS')
```

Returns 'PAS'

```
JustExtension('C:\DATAF')
```

Returns empty string ''

See Also

JustName JustPathname

Declaration

```
function JustFilename(PathName : string) : string;
```

Purpose

Return just the filename and extension of a pathname.

Description

Drive letter and directory are stripped from the input string. Pathname is generally expected to be a complete pathname, with both a path and a filename. If there is no ':' or '\' in PathName, the input string will be returned unchanged.

Examples

```
JustFilename('C:\AUTOEXEC.BAT')
```

Returns 'AUTOEXEC.BAT'

```
JustFilename('test.')
```

Returns 'test.'

See Also

JustName JustPathname

Declaration
```
function JustName(PathName : string) : string;
```

Purpose
Return just the name (no extension, no path) of a pathname.

Description
The path and extension are stripped from PathName, returning just the root name of the file.

Examples
```
JustName('TEST.PAS')
```
Returns 'TEST'
```
JustName('C:\AUTOEXEC.BAT')
```
Returns 'AUTOEXEC'

Declaration
```
function JustPathname(PathName : string) : string;
```

Purpose
Return just the drive:directory portion of a pathname.

Description
The filename and extension of the input string are stripped. Pathname is generally expected to be a complete pathname, with both a path and a filename. If there is no ':' or '\' in PathName, the input string will be returned unchanged.

Examples
```
JustPathname('C:\AUTOEXEC.BAT')
```
Returns 'C:\'
```
JustPathname('..\test\bug.pas')
```
Returns '..\test'

See Also
JustExtension JustFilename

Declaration
```
function LeftPad(S : string; Len : Byte) : string;
```

Purpose
Return a string left-padded to length Len with blanks.

Description
Blanks are inserted to the left of the input string until the total length equals Len. If S is longer than Len, S is returned unchanged. If S is empty, a string of blanks is returned.

Examples
```
LeftPad('hello', 7)
```
Returns ' hello'
```
LeftPad('gargantuan', 5)
```
Returns 'gargantuan'

See Also
Center Pad

LeftPadCh / LoCase / LoCaseMac

Declaration

```
function LeftPadCh(S : string; Ch : Char; Len : Byte) : string;
```

Purpose

Return a string left-padded to length Len with Ch.

Description

Characters are inserted to the left of the input string until its length equals Len. If S is longer than Len, then S is returned unchanged. If S is the empty string, a string of Ch is returned.

Examples

```
LeftPadCh('hello', '-', 7)
```

Returns '--hello'

```
LeftPadCh('gargantuan', 5)
```

Returns 'gargantuan'

See Also

Center Pad

Declaration

```
function LoCase(Ch : Char) : Char;
```

Purpose

Return Ch in lower case.

Description

LoCase converts uppercase characters to lower case. Unlike LoCaseMac, it correctly handles all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

Examples

```
LoCase('A')
```

Returns 'a'

```
LoCase(#165) {'N' with a tilde ('~')}
```

Returns #164 ('n' with a tilde)

See Also

StLoCase UpCase

Declaration

```
function LoCaseMac(Ch : Char) : Char;
```

Purpose

Return Ch in lower case.

Description

An inline macro that converts characters to lower case as rapidly as possible.

Examples

```
LoCaseMac('A')
```

Returns 'a'

```
LoCaseMac(#165)
```

Returns #165

See Also

UpCaseMac

Declaration

```
function Long2Str(L : LongInt) : string;
```

Purpose

Convert a LongInt/Word/Integer/Byte/ShortInt to a string.

Description

This routine calls Turbo's built-in Str procedure and then assigns the result to the function. The resulting string has no leading or trailing blanks.

Turbo's automatic type promotion allows bytes, integers or words to be used as the parameter to this routine.

Examples

```
Long2Str(1000)
```

Returns '1000'

```
Long2Str(-1)
```

Returns '-1'

See Also

Str2Int

Declaration

```
function LongIntForm(Mask : string; L : LongInt) : string;
```

Purpose

Returns a formatted string with digits from L merged into the Mask

Description

LongIntForm functions in the same way as Form, except that all commas and/or periods in the Mask are treated as separators.

Examples

```
Form('@@@@@', 1234)
```

Returns '01234'

```
Form('##,###,###', 12345678)
```

Returns '12,345,678'

```
Form('##,###.###', 12345678)
```

Bad mask! LongIntForm returns '12,345.678' because '.' and ',' are both treated as separators.

See Also

Form

MakeLetterSet / OctalB / OctalL

Declaration

```
function MakeLetterSet(S : string) : LongInt;
```

Purpose

Returns a bit-mapped LongInt storing the individual letters contained in S.

Description

MakeLetterSet creates a bit-mapped set (in the form of a LongInt) indicating which letters of the alphabet are in S. For example, if the string were 'A' (or 'a'), MakeLetterSet would return $00000001; if it were 'ABC', the result would be $00000007, and so on. The maximum possible value of the result is $003FFFFF (first 26 bits set). See the discussion of CompareLetterSets for more details.

See Also

CompareLetterSets

Declaration

```
function OctalB(B : Byte) : string;
```

Purpose

Return octal string for Byte.

Description

Returns a 3-character string in octal (base 8) format. Leading zeros are included.

Examples

```
OctalB(0)
```

Returns '000'

```
OctalB(255)
```

Returns '377'

Declaration

```
function OctalL(L : LongInt) : string;
```

Purpose

Return octal string for LongInt.

Description

Returns a 12-character string in octal (base 8) format. Leading zeros are included. Examples OctalL(0) Returns '000000000000' OctalL(655360) Returns '000002400000'

See Also

OctalW

Declaration
```
function OctalW(W : Word) : string;
```

Purpose
Return octal string for Word.

Description
Returns a 6-character string in octal (base 8) format. Leading zeros are included.

Examples
```
OctalW(0)
```
Returns '000000'
```
OctalW(65535)
```
Returns '177777'

See Also
OctalB OctalL

Declaration
```
function Pad(S : string; Len : Byte) : string;
```

Purpose
Return a string right-padded to length Len with blanks.

Description
Blanks are added to the end of the input string until its length equals Len. If the length of S already equals or exceeds Len, S is returned unchanged. If S is the empty string, a string of blanks is returned.

Examples
```
Pad('hello', 8)
```
Returns 'hello '
```
Pad('gargantuan', 8)
```
Returns 'gargantuan'

See Also
Center LeftPad

Declaration
```
function PadCh(S : string; Ch : Char; Len : Byte) : string;
```

Purpose
Return a string right-padded to length Len with Ch.

Description
Characters are added to the end of the input string until its length equals Len. If the length of S already equals or exceeds Len, S is returned unchanged. If S is the empty string, a string of Ch is returned.

Examples
```
PadCh('hello', '-', 8)
```
Returns 'hello---'
```
PadCh('gargantuan', '-', 8)
```
Returns 'gargantuan'

Real2Str

Declaration

```
function Real2Str(R : Float; Width : Byte; Places : ShortInt) : string;
```

Purpose

Convert a real to a string.

Description

This routine calls Turbo's built-in Str procedure to convert R to a string of length Width, which is returned as the function result. Places is the number of decimal places to round to. If places is -1, the string returned will be in exponential format.

Note: If OPSTRING is compiled with {$N+}, this routine works with variables of type Extended, else with variables of type Real.

Examples

```
R := 1234.56789;
Str(R:7:2, S);
S := Real2Str(R, 7, 2);
```

In both cases, S would be set to '1234.57'.

```
Real2Str(1.23456, 10, -1)
```

Returns ' 1.235E+00' ($N-) or ' 1.2E+0000' ($N+)

Declaration

```
function Search(var Buffer; BufLength : Word;
                var Match; MatLength : Word) : Word;
```

Purpose

Search through Buffer for Match.

Description

This is a very fast general purpose search function that applies equally well to strings or longer byte sequences. The variable to be searched is passed in the Buffer parameter. The length of the region to be searched is passed in BufLength. The string to match is passed in Match. The number of characters to match is passed in MatLength.

If the Match string is found, Search returns the number of characters that were searched to reach the beginning of the Match string. Note that this may be zero or more characters. If the Match string is not found, the value $FFFF is returned.

Examples

```
var
  S, T : string;
  Posn : Word;
...
  S := 'Now is the time for the show to start';
  T := 'Now';
  Posn := Search(S[1], Length(S), T[1], Length(T)); {Posn is 0}
  T := 'the';
  Posn := Search(S[0], Length(S), T[1], Length(T)); {Posn is 8}
  Posn := Search(S[Posn+1], Length(S)-Posn, T[1], Length(T)); {Posn is 13}
  T := 'zzz';
  Posn := Search(S[0], Length(S), T[1], Length(T)); {Posn is $FFFF}
```

Starting the search at position 0 of the string makes Search return match values just like those of Turbo's Pos function. Starting at position zero may be dangerous though, since the length byte will be included in the characters searched. By indexing the Buffer string as shown in the example, previous matches may be skipped.

See Also

BMSearch PosAsc (OPASCIIZ)

Declaration

```
function SearchUC(var Buffer; BufLength : Word;
                  var Match; MatLength : Word) : Word;
```

Purpose

Case-insensitive search through Buffer for a Match.

Description

The function works in a manner identical to Search except that the comparison is made in a case-insensitive manner. Characters in both the Buffer and the Match strings are raised to upper case prior to comparison. The uppercasing does not change the actual parameters.

If the Match string is not found, SearchUC returns $FFFF. Otherwise it returns the number of characters searched to find the start of the match string.

SearchUC runs 2-4 times slower than Search, due to the uppercasing operation.

Note that SearchUC handles correctly all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

See Also

BMSearchUC Search

Soundex / StLoCase

Declaration

```
function Soundex(S : string) : string;
```

Purpose

Return 4-character soundex of input string.

Description

The soundex code is often used to provide approximate matching of proper names. Soundex returns a 4-character string encoding the input string with the soundex algorithm. The first character is just the uppercase value of the input string's first character. The remaining three characters are numeric digits that encode the sound of the input string. Comparing the soundex strings of two similar names will usually yield an exact match.

Soundex is useful only for strings comprised mostly of alphabetic characters. Leading blanks should be removed prior to calling Soundex. Soundex internally raises the string to upper case.

Examples

```
Soundex('macintosh')
```

Returns 'M253'

```
Soundex('Mcntoch')
```

Returns 'M253'

Declaration

```
function StLoCase(S : string) : string;
```

Purpose

Convert uppercase letters in string to lower case.

Description

Note that StLoCase handles correctly all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

Examples

```
StLoCase('ABC')
```

Returns 'abc'

```
StLoCase('!a# rats!')
```

Returns '!@# rats!'

See Also

LoCase StUpCase

Declaration

```
function Str2Int(S : string; var I : Integer) : Boolean;
```

Purpose

Convert a string to an integer, returning true if successful.

Description

This routine calls the built-in Turbo Val procedure, assigning the numeric result to the var parameter, and the success code to a boolean function result. Any trailing blanks are stripped from S before Val is called.

If Str2Int returns False, the var parameter I will contain the position within S that caused the error. If the input string begins with '$', the string is interpreted as a hexadecimal number. Leading blanks are acceptable.

Examples

```
Str2Int(' 123', I)
```

Returns True and I = 123

```
Str2Int('$00F0', I)
```

Returns True and I = 240

```
Str2Int('12z', I)
```

Returns False and I = 3

Declaration

```
function Str2Long(S : string; var I : LongInt) : Boolean;
```

Purpose

Convert a string to a LongInt, returning True if successful.

Description

This routine calls the built-in Turbo Val procedure, assigning the numeric result to the var parameter, and the success code to a boolean function result. Any trailing blanks are stripped from S before Val is called.

If Str2Long returns False, the var parameter I will contain the position within S that caused the error. If the input string begins with '$', the string is interpreted as a hexadecimal number. Leading blanks are acceptable.

Example

```
Str2Long('$A0000', I)
```

Returns True and I = 655360

See Also

Long2Str

Str2Real / Str2Word

Declaration

```
function Str2Real(S : string; var R : Float) : Boolean;
```

Purpose

Convert a string to a real, returning true if successful.

Description

This routine calls the built-in Turbo Val procedure, assigning the numeric result to the var parameter and the success code to a boolean function result. Trailing blanks are trimmed before calling Val. If Str2Real returns False, the var parameter R will contain the position within S that caused the error.

Note: If OPSTRING is compiled with {$N+}, this routine works with variables of type Extended, else with variables of type Real.

Example

```
Str2Real(' -.123 ', R);
```

Returns True and R = -0.123.

```
Str2Real('123. 4567', R);
```

Returns False and R = 5.0.

Declaration

```
function Str2Word(S : string; var I : Word) : Boolean;
```

Purpose

Convert a string to a word, returning True if successful.

Description

This routine calls the built-in Turbo Val procedure, assigning the numeric result to the var parameter, and the success code to a boolean function result. Any trailing blanks are stripped from S before Val is called.

If Str2Word returns False, the var parameter I will contain the position within S that caused the error. If the input string begins with '$', the string is interpreted as a hexadecimal number. Leading blanks are acceptable.

Examples

```
Str2Word(' 123', I)
```

Returns True and I = 123

```
Str2Word('12z', I)
```

Returns False and I = 3

See Also

Long2Str

Declaration

```
function StringFromHeap(P : StringPtr) : string;
```

Purpose

Return string at P^.

Description

Applications that perform extensive text processing are often better off storing strings on the heap rather than as global variables. The heap offers two advantages in this situation: it can grow to use all 640K of DOS memory, and strings can take just the memory they need rather than using the maximum possible length. For example, suppose a filter program must read in each line of a text file and store it in memory prior to processing. By storing each line as global data, even one character consumes the maximum line length. By storing the lines on the heap, each string consumes exactly what it needs.

StringFromHeap works in concert with StringToHeap. StringToHeap allocates heap space for a specified string and returns a pointer to the string. StringFromHeap returns the actual string from the heap for further use in an expression.

If the pointer passed to StringFromHeap is nil, the empty string is returned. StringFromHeap may be used in string expressions.

Example

```
var
  P : pointer;
  S : string;
begin
  P := StringToHeap('PASFILE.INC');
  S := 'C:\'+StringFromHeap(P);
end;
```

StringToHeap allocates 12 bytes of heap space for 'PASFILE.INC' plus its length byte. StringFromHeap loads 'C:\PASFILE.INC' into S.

See Also

DisposeString

Declaration

```
function StringToHeap(S : string) : StringPtr;
```

Purpose

Allocate space for S and return pointer.

Description

StringToHeap works in concert with StringFromHeap and DisposeString. StringToHeap allocates heap space for a specified string and returns a pointer to the string. StringFromHeap returns the actual string from the heap for further use in an expression.

If insufficient heap space remains to store a string, StringToHeap returns a nil pointer.

Example

See above.

See Also

DisposeString StringFromHeap

StUpCase / Trim / TrimLead

Declaration

```
function StUpCase(S : string) : string;
```

Purpose

Convert lowercase letters in string to upper case.

Description

StUpCase converts S to upper case, correctly handling all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

Examples

```
StUpCase('hello')
```

Returns 'HELLO'

```
StUpCase('!@#$ drat!')
```

Returns '!@#$ DRAT!'

See Also

StLoCase UpCase

Declaration

```
function Trim(S : string) : string;
```

Purpose

Return a string with leading and trailing white space removed.

Description

White space is defined as any ASCII character whose ordinal value is 32 or less. White space internal to the string S is returned unchanged. If the input string is composed entirely of white space, the empty string is returned.

Examples

```
Trim('  hello  ')
```

Returns 'hello'

```
Trim(^I^I'hello'^M^J)
```

Returns 'hello'

Declaration

```
function TrimLead(S : string) : string;
```

Purpose

Return a string with leading white space removed.

Description

White space is defined as any ASCII character whose ordinal value is 32 or less. White space internal to the string S is returned unchanged. If the input string is composed entirely of white space, the empty string is returned.

Examples

```
TrimLead('  hello  ')
```

Returns 'hello '

```
TrimLead(^I^I'hello'^M^J)
```

Returns 'hello'^M^J

Declaration

```
function TrimSpaces(S : string) : string;
```

Purpose

Return a string with leading and trailing spaces removed.

Description

TrimSpaces is identical to Trim, except that it trims only spaces. Control characters (#0-#31) are left unchanged.

Examples

```
TrimSpaces('  hello  ')
```

Returns 'hello'

```
TrimSpaces(^I^I'hello'^M^J)
```

Returns ^I^I'hello'^M^J

Declaration

```
function TrimTrail(S : string) : string;
```

Purpose

Return a string with trailing white space removed.

Description

White space is defined as any ASCII character whose ordinal value is 32 or less. White space internal to the string S is returned unchanged. If the input string is composed entirely of white space, the empty string is returned.

Examples

```
TrimTrail('  hello  ')
```

Returns ' hello'

```
TrimTrail(^I^I'hello'^M^J)
```

Returns ^I^I'hello'

See Also

Trim TrimLead

UpCase / UpCaseMac

Declaration

```
function UpCase(Ch : Char) : Char;
```

Purpose

Return Ch in upper case.

Description

This routine converts characters to upper case. Unlike both the standard UpCase function in SYSTEM and UpCaseMac, UpCase correctly handles all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

Examples

```
UpCase('a')
```

Returns 'A'

```
UpCase(#164) {'n' with a tilde ('~')}
```

Returns #165 ('N' with a tilde)

See Also

LoCase StUpCase

Declaration

```
function UpCaseMac(Ch : Char) : Char;
```

Purpose

Return Ch in upper case.

Description

An inline macro that converts characters to upper case as rapidly as possible.

Examples

```
UpCaseMac('a')
```

Returns 'A'

```
UpCaseMac(#164)
```

Returns #164

See Also

LoCaseMac

Declaration

```
function WordCount(S : string; WordDelims : CharSet) : Byte;
```

Purpose

Given a set of word delimiters, return number of words in S.

Description

This routine returns the number of words in the specified string. WordDelims is a user-defined set of word delimiters used to determine where words begin and end. At the very least, this set should contain ' ', but in some applications you may want to use a larger set like the one in the example below.

Example

```
var
  S : string;
const
  WordDelims : CharSet =
    [#0..'"', '.', ',', ':', ';', '?', '(', ')', '[', ']', '{', '}'];
  N : Word = 0;
begin
  while not EOF(Input) do begin
    ReadLn(Input, S);
    Inc(N, WordCount(S, WordDelims));
  end;
  WriteLn(N, ' words in input file');
end.
```

A simple word counting program that reads from the standard input device.

See Also

ExtractWord

Declaration

```
function WordPosition(N : Byte; S : string; WordDelims : CharSet) : Byte;
```

Purpose

Given a set of word delimiters, return start position of N'th word in S.

Description

This routine returns the position of the N'th word in string S. WordDelims is a user-defined set of word delimiters used to determine where words begin and end. At the very least, this set should contain ' '. If S contains fewer than N words, WordPosition returns 0.

Example

```
S := 'Four score and seven years ago, our forefathers...';
I := WordPosition(7, S, [' ', ',', '.']);
```

The result would be 33 (the position of the 'o' in our').

WordWrap

Declaration

```
procedure WordWrap(InSt : string; var OutSt, Overlap : string;
                   Margin : Byte; PadToMargin : Boolean);
```

Purpose

Wrap InSt at Margin, storing the result in OutSt and the remainder in Overlap.

Description

This routine takes as input a string of text (InSt) and returns an output string (OutSt) no greater than Margin characters in length. If the margin falls in the middle of a word (defined as a series of characters bounded by either spaces or one end of the string), the output string will contain all previous words in the string, while all text from the word straddling the margin to the end of the string will be returned in Overlap. OutSt must therefore be large enough to hold a string of Margin characters, and Overlap must be at least as large as InSt.

If the input string cannot be word wrapped--that is, if Length(InSt) is greater than Margin, and there are no spaces between InSt[1] and InSt[Margin]--the output string will simply contain the first Margin characters in InSt.

PadToMargin indicates whether or not the output string should be padded with blanks to the specified Margin.

Example

```
var
  InputString, OutputString : string;
begin
  while not EOF(Input) do begin
    ReadLn(Input, InputString);
    InputString := Detab(InputString, 8);
    repeat
      WordWrap(InputString, OutputString, InputString, 40, False);
      WriteLn(Output, OutputString);
    until Length(InputString) = 0;
  end;
end.
```

A simple filter program for ASCII text files that expands all tabs in input lines and wraps output lines at column 40.

See Also

Detab

C. OPASCIIZ: Long ASCIIZ Strings

The OPASCIIZ unit defines a new string type for Turbo Pascal. Unlike normal Turbo strings, which are limited to 255 bytes in length, the ASCIIZ string type can be of any length up to 65,520 characters.

The ASCIIZ string is defined in the same way as the one used by MS-DOS for internal operations and by the C programming language. An ASCIIZ is simply an array of characters terminated by a null character (ASCII #0).

Two disadvantages arise when using ASCIIZ strings. First, the length of such a string cannot be immediately determined. Instead, a length routine must scan the entire string until it encounters a null. OPASCIIZ implements this scan with a Turbo inline macro, and uses the 8088 SCASB instruction to obtain best possible performance for this operation. As a result, performance of the length operation should not be a practical problem.

The second disadvantage may rule out the use of ASCIIZ strings for some applications. Because a null character terminates an ASCIIZ, null characters may not be part of the string itself. Thus, ASCIIZ strings cannot contain extended scan code sequences (which begin with a null) unless the nulls are first translated into another character.

Although ASCIIZ strings may be as long as 65,520 characters, the OPASCIIZ unit defines a default maximum length of 1024 characters. This limit will allow you to declare a number of ASCIIZ strings as global variables without gobbling up the entire 64K segment. If you need even longer strings, change the constant MaxAsciiz. In this case, ASCIIZ strings should probably be allocated on the heap.

Constants

```
MaxAsciiz = 1024;
```

The maximum length of an Asciiz string. It may be increased up to 65,520.

```
NotFound = $FFFF;
```

Value returned by PosAsc and PosStr if the target substring is not found.

Types

```
Asciiz = array[0..MaxAsciiz] of Char;
```

The internal representation of an ASCIIZ string. One extra character position is allocated to store the null termination byte.

```
AsciizPtr = ^Asciiz;
```

A pointer to an ASCIIZ string. Because these strings are so large, it is often desirable to allocate them on the heap.

Asc2Str / AscCenter / AscCenterCh

Declaration

```
function Asc2Str(var A : Asciiz) : string;
```

Purpose

Convert Asciiz to Turbo string.

Description

If the Asciiz exceeds 255 characters, it is truncated prior to assignment to the Turbo string.

Example

```
var A : Asciiz;
    S : string;
...
Str2Asc('hello', A);
S := Asc2Str(A); {S gets 'hello'}
```

See Also

Str2Asc

Declaration

```
procedure AscCenter(var A : Asciiz; Width : Word; var B : Asciiz);
```

Purpose

Return an Asciiz centered in an Asciiz of blanks with specified width.

Description

If A already exceeds Width, the returned Asciiz B is the same as A. Otherwise, B returns an Asciiz of specified Width with A centered in a field of blanks.

Example

```
var A, B : Asciiz;
...
Str2Asc('hello', A);
AscCenter(A, 10, B); {B gets '   hello  '}
```

See Also

Center (OPSTRING)

Declaration

```
procedure AscCenterCh(var A : Asciiz; Ch : Char; Width : Word; var B : Asciiz);
```

Purpose

Return an Asciiz centered in an Asciiz of Ch with specified width.

Description

If A already exceeds Width, the returned Asciiz B is the same as A. Otherwise, B returns an Asciiz of specified Width with A centered in a field of Ch.

Example

```
var A, B : Asciiz;
...
Str2Asc('hello', A);
AscCenterCh(A, '-', 10, B); {B gets '---hello--'}
```

See Also

CenterCh (OPSTRING)

Declaration

```
procedure AscCharStr(Ch : Char; Len : Word; var A : Asciiz);
```

Purpose

Return an Asciiz of length Len filled with Ch.

Description

If Len is zero, a null Asciiz is returned. Otherwise an Asciiz of length Len is returned, filled with character Ch.

Example

```
AscCharStr('-', 10, A)
```

Leaves A as '----------'

See Also

CharStr (OPSTRING)

Declaration

```
procedure AscFromHeap(P : AsciizPtr; var A : Asciiz);
```

Purpose

Return an Asciiz from the heap, empty if pointer is nil.

Description

Because Asciiz strings can be so long, it is often desirable to store them on the heap rather than in the data segment where they can quickly consume the available 64K bytes. AscFromHeap works in conjunction with AscToHeap and DisposeAsc.

AscToHeap allocates heap space for an Asciiz and returns a pointer to it. AscFromHeap returns the data from the heap to another variable. DisposeAsc deallocates the heap space when it is no longer needed.

If the pointer P is nil, AscFromHeap returns an empty string.

Note that the statement

```
A := P^;
```

yields the same result as

```
AscFromHeap(P, A);
```

However, the call to AscFromHeap may be faster since it moves only the actual data rather than the total number of bytes in the structure.

Example

```
var
  P : AsciizPtr;
  A : Asciiz;
begin
  Str2Asc('hello', A);
  P := AscToHeap(A);
  AscFromHeap(P, A); {A gets 'hello'}
end.
```

See Also

AscToHeap
DisposeAsc
StrFromHeap (OPSTRING)

AscLeftPad / AscLeftPadCh / AscLoCase

Declaration

```
procedure AscLeftPad(var A : Asciiz; Len : Word; var B : Asciiz);
```

Purpose

Left-pad the Asciiz in A to length Len with blanks, returning the result in B.

Description

If the length of A already matches or exceeds Len, then B returns the same as A. Otherwise, A is transferred to B and then blanks are added to the start of B until the length reaches Len.

Example

```
var A, B : Asciiz;
...
Str2Asc('hello', A);
AscLeftPad(A, 10, B); {B gets '     hello'}
```

See Also

AscPad LeftPad (OPSTRING)

Declaration

```
procedure AscLeftPadCh(var A : Asciiz; Ch : Char; Len : Word; var B : Asciiz);
```

Purpose

Left-pad the Asciiz in A to length Len with Ch, returning the result in B.

Description

If the length of A matches or exceeds Len, B returns the same as A. Otherwise, A is transferred to B and then characters Ch are added to the start of B until the length reaches Len.

Example

```
var A, B : Asciiz;
...
Str2Asc('hello', A);
AscLeftPadCh(A, '-', 10, B); {B gets '-----hello'}
```

See Also

AscPadCh LeftPadCh (OPSTRING)

Declaration

```
procedure AscLoCase(var A, B : Asciiz);
```

Purpose

Convert the Asciiz in A to lower case, returning the result in B.

Description

Each character of A is set to lower case and the result is returned in B. Note that AscLoCase handles correctly all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

Example

```
var A, B : Asciiz;
...
Str2Asc('HELLO', A);
AscLoCase(A, B); {B gets 'hello'}
```

See Also

StLoCase (OPSTRING)

Declaration

```
procedure AscPad(var A : Asciiz; Len : Word; var B : Asciiz);
```

Purpose

Right-pad the Asciiz in A to length Len with blanks, returning the result in B.

Description

If the length of A already exceeds Len, then B is set to A. Otherwise, B is set to A and then blanks are added to the end of B until its length reaches Len.

Example

```
var A, B : Asciiz;
...
Str2Asc('hello', A);
AscPad(A, 10, B); {B gets 'hello
```

See Also

AscLeftPad PadCh (OPSTRING)

Declaration

```
procedure AscPadCh(var A : Asciiz; Ch : Char; Len : Word; var B : Asciiz);
```

Purpose

Right-pad the Asciiz in A to length Len with Ch, returning the result in B.

Description

If the length of A already exceeds Len, then B is set to A. Otherwise, B is set to A and then characters Ch are added to the end of B until its length reaches Len.

Example

```
var A, B : Asciiz;
...
Str2Asc('hello', A);
AscPadCh(A, '-', 10, B); {B gets 'hello-----'}
```

See Also

AscLeftPadCh PadCh (OPSTRING)

AscToHeap / AscTrim

Declaration

```
function AscToHeap(var A : Asciiz) : AsciizPtr;
```

Purpose

Put Asciiz on heap, returning a pointer, nil if insufficient memory.

Description

Because Asciiz strings can be so long, it is often desirable to store them on the heap rather than in the data segment where they can quickly consume the available 64K bytes. AscToHeap works in conjunction with AscFromHeap and DisposeAsc.

AscToHeap allocates heap space for an Asciiz and returns a pointer to it. AscFromHeap returns the data from the heap to another variable. DisposeAsc deallocates the heap space when it is no longer needed.

If sufficient heap space is not available, AscToHeap returns a nil pointer. Note that AscToHeap allocates only the exact amount of space needed for the Asciiz passed as its parameter. If instead you want to allocate enough space for the longest Asciiz, use the following statement:

```
New(P); {where P is of type AsciizPtr}
```

Example

```
var
  P : AsciizPtr;
  A : Asciiz;
begin
  Str2Asc('hello', A);
  P := AscToHeap(A); {P^ holds 'hello'#0}
end.
```

See Also

AscFromHeap
DisposeAsc
StrFromHeap (OPSTRING)

Declaration

```
procedure AscTrim(var A, B : Asciiz);
```

Purpose

Return an Asciiz with leading and trailing white space removed.

Description

White space is defined as any ASCII character whose ordinal value is 32 or less, with the exception of #0, which is used as the Asciiz termination character. If the Asciiz A consists entirely of white space, a null string is returned in B. White space internal to A is passed along unchanged.

Example

```
var A, B : Asciiz;
...
Str2Asc(' hello world! ', A);
AscTrim(A, B); {B gets 'hello world!'}
```

See Also

AscTrimLead
AscTrimTrail
Trim (OPSTRING)

Declaration

```
procedure AscTrimLead(var A, B : Asciiz);
```

Purpose

Return an Asciiz with leading white space removed.

Description

White space is defined as any ASCII character whose ordinal value is 32 or less except #0, which is used as the Asciiz termination character. If the Asciiz A consists entirely of white space, a null string is returned in B. White space internal to A is passed along unchanged.

Example

```
var A, B : Asciiz;
...
Str2Asc(' hello world! ', A);
AscTrimLead(A, B); {B gets 'hello world! '}
```

See Also

AscTrim TrimLead (OPSTRING)

Declaration

```
procedure AscTrimTrail(var A, B : Asciiz);
```

Purpose

Return an Asciiz with trailing white space removed.

Description

White space is defined as any ASCII character whose ordinal value is 32 or less except #0, which is used as the Asciiz termination character. If the Asciiz A consists entirely of white space, a null string is returned in B. White space internal to A is passed along unchanged.

Example

```
var A, B : Asciiz;
...
Str2Asc(' hello world! ', A);
AscTrimTrail(A, B); {B gets ' hello world!'}
```

See Also

AscTrimLead TrimTrail (OPSTRING)

Declaration

```
procedure AscUpCase(var A, B : Asciiz);
```

Purpose

Convert the Asciiz in A to upper case, returning the result in B.

Description

B gets A with lowercase characters converted to upper case. AscUpCase handles correctly all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

Example

```
var A, B : Asciiz;
...
Str2Asc('hello!', A);
AscUpCase(A, B); {B gets 'HELLO!'}
```

See Also

AscLoCase StUpCase (OPSTRING)

CompAsc / CompUCAsc / ConcatAsc

Declaration
```
function CompAsc(var A1, A2 : Asciiz) : AscCompareType;
```

Purpose
Return less, equal, greater if A1<A2, A1=A2, or A1>A2.

Description
Two Asciiz strings are compared using the same rules as for normal strings. The function returns an enumerated type taking the values AscLess, AscEqual, and AscGreater.

Examples
See the entry for CompString (OPSTRING) for examples.

See Also
CompString (OPSTRING)

Declaration
```
function CompUCAsc(var a1, a2 : Asciiz) : AscCompareType;
```

Purpose
Compare two Asciiz's in a case-insensitive manner.

Description
Two Asciiz strings are compared using the same rules as for normal strings. During comparison, all lowercase characters are converted to upper case, without affecting the actual parameters. The function returns an enumerated type taking the values AscLess, AscEqual, and AscGreater. Note that CompUCAsc handles correctly all alphabetics in the international character set ('A'..'Z', 'a'..'z', #128..#154, #160..#167).

Examples
See the entry for CompUCString (OPSTRING) for examples.

See Also
CompUCString (OPSTRING)

Declaration
```
procedure ConcatAsc(var A, B, C : Asciiz);
```

Purpose
Concatenate two Asciiz strings, returning a third.

Description
This procedure is analogous to the normal string expression C := A+B. If the resulting length exceeds the maximum for Asciiz strings, the result is truncated.

Example
```
var A, B, C : Asciiz;
...
Str2Asc('hello ', A);
Str2Asc('world!', B);
ConcatAsc(A, B, C); {C gets 'hello world!'}
```

See Also
ConcatStr

Declaration

```
procedure ConcatStr(var A : Asciiz; S : string; var C : Asciiz);
```

Purpose

Concatenate a string to an Asciiz.

Description

This procedure is analogous to the normal string expression C := A+S. In this case a normal Turbo string S is concatenated to the Asciiz A to return the Asciiz C. If the resulting Asciiz exceeds the maximum length, it is truncated.

Example

```
var A, B : Asciiz;
...
Str2Asc('hello ', A);
ConcatStr(A, 'world!', B); {B gets 'hello world!'}
```

See Also

ConcatAsc

Declaration

```
procedure CopyAsc(var A : Asciiz; Start, Len : Word; var O : Asciiz);
```

Purpose

Return a substring of A.

Description

This procedure is analogous to the Copy function for normal strings. Len characters are taken from position Start of A and returned in O. Note that Start = 0 refers to the first character of A.

If more characters are requested from A than exist, those that exist are returned. If the Start position exceeds the length of A, the null string is returned.

Example

```
var A, B : Asciiz;
...
Str2Asc('hello ', A);
CopyAsc(A, 1, 3, B); {B gets 'ell'}
CopyAsc(A, 4, 10, B); {B gets 'o'}
```

DeleteAsc / DisposeAsc

Declaration

```
procedure DeleteAsc(var A : Asciiz; Start, Len : Word);
```

Purpose

Delete Len characters of A, starting at position Start.

Description

This procedure is analogous to the Delete function for normal strings. Len characters are removed from position Start of A. Note that Start = 0 refers to the first character of A.

If the deletion calls for more characters from A than exist, then the existing ones are deleted. If the Start position exceeds the length of A, then A is not changed.

Example

```
var A : Asciiz;
...
Str2Asc('hello ', A);
DeleteAsc(A, 10, 10); {A is not changed}
DeleteAsc(A, 1, 1); {A gets 'hllo'}
DeleteAsc(A, 0, 2); {A gets 'llo'}
```

Declaration

```
procedure DisposeAsc(P : AsciizPtr);
```

Purpose

Dispose of heap space pointed to by P.

Description

Because Asciiz strings can be so long, it is often desirable to store them on the heap rather than in the data segment where they can quickly consume the available 64K bytes. DisposeAsc works in conjunction with AscToHeap.

AscToHeap allocates heap space for an Asciiz and returns a pointer to it. DisposeAsc deallocates the heap space when it is no longer needed.

If the pointer P is nil, DisposeAsc does nothing. DisposeAsc should not be used to dispose of pointers allocated directly with a New statement. In those cases, use the built-in Turbo Dispose procedure to deallocate the heap space.

Example

```
var
  P : AsciizPtr;
  A : Asciiz;
begin
  Str2Asc('hello', A);
  P := AscToHeap(A);
  DisposeAsc(P); {6 bytes of heap space freed}
end.
```

See Also

AscToHeap

DisposeString (OPSTRING)

Declaration

```
procedure InsertAsc(var Obj, A : Asciiz; Start : Word);
```

Purpose

Insert Asciiz Obj at position Start of A.

Description

This routine is analogous to the Insert procedure for normal strings. The Obj string is inserted at position Start of the Asciiz A. Note that Start = 0 refers to the first position in A.

If inserting Obj causes A to exceed the maximum length for Asciiz strings, A is truncated. If Start is a position beyond the end of A, Obj is concatenated to A.

Example

```
var A, B : Asciiz;
...
Str2Asc('hello ', A);
Str2Asc('world!', B);
InsertAsc(A, B, 0); {B gets 'hello world!'}
```

Declaration

```
procedure InsertStr(Obj : string; var A : Asciiz; Start : Word);
```

Purpose

Insert string Obj at position Start of A.

Description

This routine is analogous to the Insert procedure for normal strings. Normal Turbo string Obj is inserted at position Start of the Asciiz A. Note that Start = 0 refers to the first position in A.

If inserting Obj causes A to exceed the maximum length for Asciiz strings, A is truncated. If Start is a position beyond the end of A, Obj is concatenated to A.

Example

```
var A : Asciiz;
...
Str2Asc('world!', A);
InsertStr('hello ', A, 0); {B gets 'hello world!'}
```

LenAsc / PosAsc

Declaration

```
function LenAsc(A : Asciiz) : Word;
```

Purpose

Return the length of an Asciiz string.

Description

The length of an Asciiz string is the count of all of its characters, not including the trailing null that terminates the string. Thus, the length may range from 0 to MaxAsciiz.

Because LenAsc performs a counting operation, it is implemented as an assembly language inline macro. As such, it consumes 12-14 bytes more code space per call than would an actual call to a function. Moreover, because each scan for the end of a string takes time, LenAsc should be called a minimum number of times by assigning its result to a word type variable.

Examples

```
var A : Asciiz;  L : word;
...
Str2Asc('hello', A);
L := LenAsc(A); {L gets 5}
DeleteAsc(A,0,5);
L := LenAsc(A); {L gets 0}
```

Declaration

```
function PosAsc(var Obja, A : Asciiz) : Word;
```

Purpose

Return the position of ObjA in A.

Description

The PosAsc routine is analogous to the Pos function for normal Turbo strings. It returns the position of ObjA in A. Note that the first position of A is 0. If ObjA is not found in A, then PosAsc returns $FFFF (which may be referred to using the defined constant NotFound).

Examples

```
var A, B : Asciiz;  Posn : word;
...
Str2Asc('hello world!', A);
Str2Asc('aardvark', B);
Posn := PosAsc(B, A); {Posn gets $FFFF}
Str2Asc('hello', B);
Posn := PosAsc(B, A); {Posn gets 0}
```

See Also

PosStr Search (OPSTRING)

Declaration

```
function PosStr(Obj : string; var A : Asciiz) : Word;
```

Purpose

Return the position of the string Obj in A, or NotFound if not found.

Description

The PosStr routine is analogous to the Pos function for normal Turbo strings. It returns the position of the normal Turbo string Obj in the Asciiz A. Note that the first position of A is 0. If Obj is not found in A, then PosStr returns $FFFF (which may be referred to using the defined constant NotFound).

Examples

```
var
  A : Asciiz;
  Posn : word;
begin
  Str2Asc('hello world!', A);
  Posn := PosStr('aardvark', A); {Posn gets $FFFF}
  Posn := PosStr('hello', A); {Posn gets 0}
end.
```

See Also

PosAsc Search (OPSTRING)

Declaration

```
function ReadLnAsc(var F : Text; var A : Asciiz) : Boolean;
```

Purpose

Read an Asciiz from a text file, returning True if successful.

Description

This function lets you read long strings from text files. F is a text file previously opened for reading with any buffer size. The parameter A will return an Asciiz string from the file, ending at the next carriage return / line feed combination (ASCII #13#10).

If an I/O error occurs, ReadLnAsc returns False. If F is already at end of file, ReadLnAsc returns False and A is the null string. If the line of text exceeds the maximum length for an Asciiz, the remainder will be truncated.

Example

```
var
  F : text;
  A : Asciiz;
begin
  Assign(F, 'LONG.TXT');
  Reset(F);                          {open for reading}
  while not eof(F) do begin          {read while not end of file}
    if not ReadLnAsc(F, A) then      {read the next line}
      halt;                          {halt on error}
    if not WriteAsc(Output, A) then {write the Asciiz to screen}
      halt;                          {halt on error}
    WriteLn;                         {add end of line}
  end;
end.
```

See Also

WriteAsc

Str2Asc / WriteAsc

Declaration

```
procedure Str2Asc(S : string; var A : Asciiz);
```

Purpose

Convert a Turbo string into an Asciiz.

Description

This procedure provides a convenient way to initialize Asciiz strings. It converts a normal Turbo string (up to 255 characters long) into an Asciiz. The procedure may also be handy when interfacing Turbo procedures to various MS-DOS functions that call for Asciiz strings. Be warned that any null character (ASCII #0) embedded in the Turbo string will cause a premature termination of the Asciiz.

Example

```
var A : Asciiz;
...
Str2Asc('hello', A); {A gets 'hello'#0}
```

See Also

Asc2Str

Declaration

```
function WriteAsc(var F : Text; var A : Asciiz) : Boolean;
```

Purpose

Write an Asciiz to a text file, returning True if successful.

Description

This routine lets you write long strings to text files. F is any Turbo text file previously opened for writing. The Asciiz A is written to the file without its terminating null. If any I/O error occurs, the function returns False.

Example

```
var F : text;
    A : Asciiz;
begin
  Assign(F, 'LONG.TXT'); Reset(F);     {open for reading}
  while not eof(F) do begin            {read while not end of file}
    if not ReadLnAsc(F, A) then halt; {read the next line}
    WriteAsc(Output, A);               {write the Asciiz to screen}
    WriteLn;                           {write a carriage return}
  end;
end.
```

See Also

ReadLnAsc

D. OPSTRDEV: Formatted Strings

As indicated earlier, OPSTRDEV is a very small unit. It exports only a single global variable (TpStr) and two simple routines (ReadStr and ReturnStr) to the outside world. Its simplicity is a bit deceptive, however. OPSTRDEV takes advantage of one of the most powerful features of Turbo Pascal, the "text file device driver," to allow you to use all the formatting capabilities of the Write statement without sending output to a file or device.

Using OPSTRDEV is very simple: all you have to do is add it to the USES clause at the start of your program. The OPSTRDEV unit will initialize itself automatically at the start of the program, making the TpStr "device" look as if it were part of the language. To create formatted strings with it, you just use a normal Write statement. For example:

```
Write(TpStr, 'Elapsed time: ', Hours:2, ' hours, ', Minutes:2, ' minutes.');
```

You can then retrieve the formatted string in one of two ways. You can call ReadStr to load the string into a variable, or you can call ReturnStr:

```
ReadStr(MyString);
MyString := ReturnStr;
```

ReadStr is more efficient than ReturnStr (in terms of speed and stack usage) when all you need to do is put the string in a variable. ReturnStr is more flexible, since you can use the string it returns as an argument in a procedure or function call.

Variables

```
TpStr : Text;
```

This is the 'device' that you write to to create formatted strings using OPSTRDEV. For example,

```
Write(TpStr, Pi:0:2);
```

would create a string containing '3.14'. (The name, incidentally, comes from "TPSTRDEV," the predecessor of OPSTRDEV.)

ReadStr / ReturnStr

Declaration

```
procedure ReadStr(var S : string);
```

Purpose

'Read' a string into S from our string buffer.

Description

ReadStr loads the current contents of the string buffer associated with TpStr into a string variable and empties the buffer. In cases where the result is not needed immediately, it is faster and more efficient than ReturnStr.

Example

```
Real1 := 6.95;
Write(TpStr, 'Rate: $', Real1:0:2, ' per hour.');
ReadStr(MyString);
```

MyString would contain 'Rate: $ 6.95 per hour.' after the call to ReadStr.

Declaration

```
function ReturnStr : String;
```

Purpose

Return the contents of our string buffer.

Description

ReturnStr returns the current contents of the string buffer associated with TpStr and empties the buffer. Unlike ReadStr, it can be used to pass the contents of the string buffer to another procedure or function without using an intermediate string variable.

Example

```
Real1 := 6.95;
Write(TpStr, 'Rate: $', Real1:0:2, ' per hour.');
FastWrite(ReturnStr, 1, 1, TextAttr);
```

FastWrite would be passed a string containing 'Rate: $ 6.95 per hour.'.

10. DOS/BIOS Calls

A. Overview

Turbo Pascal has long given programmers easy access to DOS and the BIOS in general, but for the most part it has required you to write your own routines to perform specific tasks. The DOS unit, a standard part of the Turbo Pascal runtime library, eases the programmer's burden to a considerable degree, but even it leaves a large number of gaps to be filled. The OPDOS unit in **Object Professional** aims to fill most of these gaps. Unlike the OPCRT unit, which is a plug-in replacement for Turbo's own CRT unit, OPDOS simply complements the DOS unit. When combined, these two units contain most of the routines you should ever need to harness the power of DOS and the BIOS.

The primary emphasis in OPDOS is on file handling and disk drive access. Among other things you'll find routines to: count the number of installed drives; select a new drive; classify a drive; read and write absolute disk sectors; get and set the current Disk Transfer Address (DTA); test for the existence of a file, either in the current directory or anywhere on DOS's PATH; and perform random access I/O with text files.

A second group of routines gives you access to the DOS 3.x concurrent print facility. Once you've determined that PRINT.COM is installed by calling the PrintInstalled function, you can send a file to the print spooler simply by calling SubmitPrintFile, and you can stop the print job just as easily by calling either CancelPrintFile or CancelAllPrintFiles. You can even examine the list of files waiting to be printed by calling GetPrintStatus.

Another, still larger group of routines gives you nearly complete control over the DOS environment. You can modify the current program's copy of the environment, create a customized environment in order to pass information to a child process--even modify DOS's master copy of the environment.

Perhaps the most unusual routine in OPDOS is ExecDos. Although the Exec routine in Turbo's DOS unit is indeed useful, it presents you with two problems. First, it requires you in many cases to locate COMMAND.COM by searching through the DOS environment for the current COMSPEC setting; if you don't know how to find COMMAND.COM, you can't "shell to DOS" using Exec. Second, it forces you to determine at compile time the maximum amount of heap space your program might require; if you set the figure high to give yourself plenty of elbow room, you can radically limit the number of things that can be done with an Exec call.

ExecDos deals with both of these problems. When appropriate, it automatically locates COMMAND.COM for you. All you have to do is pass it the name of the DOS command to be executed, and a boolean flag indicating whether or not you've given it the complete pathname of an executable file. More importantly, ExecDos allows you to allocate all available memory for your program without sacrifice. Before executing your DOS command, ExecDos moves Turbo's free list down in memory and returns all memory above the current top of the heap to DOS for its own use. After the command has been executed, all the memory given to DOS is reallocated and the free list is restored to its previous position, so that your program will once again have access to all available memory.

A related routine, ShellWithPrompt, adds another useful capability. By using ExecDos together with several of the routines that manipulate the DOS environment, ShellWithPrompt allows you to create a DOS shell that displays a special DOS prompt. For example,

```
Status := ShellWithPrompt('Enter "EXIT" to return to MYPROG'^M^J, nil);
```

would display a reminder that the user needs to type "EXIT" at the DOS prompt in order to return to your program (MYPROG).

As good as ExecDos and ShellWithPrompt are, though, the Best of Class honors surely go to the ExecDosSwap routine in OPEXEC. Whereas the routines in OPDOS return to DOS all memory not currently in use by your program, ExecDosSwap can give DOS nearly all the memory allocated to your program. The complicated process used to accomplish this feat is explained in a later section. For now, suffice it to say that ExecDosSwap would allow a program that uses 450K of memory to execute a second program that uses 447K–without terminating the first program.

The OPENHKBD unit is grouped with the OPDOS unit because its main function is to interact with the BIOS. Like OPEXEC, OPENHKBD does not provide an extensive list of procedures and functions for you to call. Instead, it invisibly installs a pair of interrupt handlers that trap all system calls that request keyboard input. Their purpose is to do two things: to make it easy for you to write programs that support enhanced keyboards (the newer ones with the 12 function keys across the top), and to allow older keyboards to emulate the enhanced keyboards wherever possible.

B. OPDOS: Assorted DOS/BIOS Calls

This section describes all of the types, variables, constants, procedures and functions found in the interface section of the OPDOS unit.

Types

```
ActionCodeType = ExecSaveScreen..ExecRestoreScreen;
```

An ExecDos action code. See the entries for ExecSaveScreen, ExecShowMemory, ExecPauseAfterRun, and ExecRestoreScreen, below.

```
DiskClass = (Floppy360, Floppy720, Floppy12, Floppy144, OtherFloppy, Bernoulli,
             HardDisk, RamDisk, SubstDrive, UnknownDisk, InvalidDrive);
```

This enumerated type defines the eight classes of disks that can be identified by GetDiskClass, as well as several types indicating that a particular error occurred. The meaning of each value is described briefly below:

```
Floppy360     A 360K, 5 1/4 floppy disk
Floppy720     A 720K, 3 1/2 floppy disk
Floppy12      A 1.2 meg, 5 1/4 floppy disk
Floppy144     A 1.44 meg, 3 1/2 floppy disk
OtherFloppy   Another kind of floppy disk (e.g. a single-sided disk)
Bernoulli     A Bernoulli drive
HardDisk      Any hard disk
RamDisk       A RAM disk created with VDISK.SYS,RAMDRIVE.SYS, etc.
SubstDrive    A logical drive created by SUBST or ASSIGN
UnknownDisk   A drive that could not be classified
InvalidDrive  Indicates that the drive does not exist or isn't ready
```

```
EnvRec =
  record
    EnvSeg : Word;              {Segment of the environment}
    EnvLen : Word;              {Usable length of the environment}
    EnvPtr : Pointer;           {Nil except when allocated on heap}
  end;
```

This type is used to package information about the DOS environment. EnvSeg is the segment address of the environment. At this address starts a list of null-terminated strings that may extend for up to 32K characters. EnvLen is the maximum usable length of that environment (not the amount currently in use). EnvPtr is normally nil. It is used only when you allocate new environment space from the Turbo Pascal heap, and even then you won't want to refer to the EnvPtr field directly.

If an error occurs in any routine that works with EnvRec's, the EnvSeg field will be set to zero. You shouldn't continue processing with an environment record if EnvSeg is zero.

```
ExecDosProc = procedure(ActionCode : ActionCodeType; Param : Word);
```

A user-written routine that performs various actions during a call to ExecDos or ShellWithPrompt.

Constants

```
ExecSaveScreen    = 0;
ExecShowMemory    = 1;
ExecPauseAfterRun = 2;
ExecRestoreScreen = 3;
```

Action codes passed by ExecDos (or ShellWithPrompt) to a routine of type ExecDosProc. The ExecSaveScreen code is passed first, at an opportune time for saving the screen (using SaveWindow, for example). Note that the screen buffer to be used must have been allocated before the call to ExecDos. The ExecShowMemory code is passed just before DOS is called, when shelling to DOS (i.e., when the first parameter to ExecDos is ''). At the same time, a second parameter (a Word) is passed, indicating the approximate amount of memory available (in kilobytes). The ExecPauseAfterRun code is passed when the Exec call returns.

You may want to use this opportunity to ask the user to "Press any key...", to avoid wiping out the contents of the screen before the user can see them. The ExecRestoreScreen code is passed immediately after that, giving you a chance to restore the screen.

Note: The ExecShowMemory action code is only passed to the ExecDosProc when the ExecDos Command is blank; the ExecPauseAfterRun action code is only passed when Command is not blank.

```
MinSpaceForDos : Word = 20000;
```

Minimum number of bytes for DOS shell to run. If ExecDos cannot free up this much space for DOS before the call, it will abort.

```
StackSafetyMargin : Word = 1000;
```

Used by ExecDos when moving the free list downward in memory. If SizeOf(free list) + StackSafetyMargin < = SP (current stack pointer), the free list will be stored at the bottom of the stack, otherwise on the heap.

```
StdInHandle   = 0;
StdOutHandle  = 1;
StdErrHandle  = 2;
StdPrnHandle  = 4;
```

Standard DOS file handles (standard input, standard output, standard error output, standard printer).

Declaration
```
procedure CancelAllPrintFiles;
```

Purpose
Cancels all files in the print queue.

Description
All printing is stopped immediately. PRINT will not finish printing the current file.

Declaration
```
procedure CancelPrintFile(FileMask : string);
```

Purpose
Cancels all the files in the print queue matched by the file mask.

Description
This routine lets you remove individual files from the queue. FileMask is any valid DOS file mask, and may include wildcard characters ('?' and '*'). If the file being printed is matched by the mask, DOS will stop printing it immediately and go on to the next file in the queue.

Examples
```
CancelPrintFile('*.DOC');
```
Cancel the printing of all files with an extension of 'DOC'.
```
if SubmitPrintFile(CurrentPrintFile) <> 0 then {error};
...
CancelPrintFile(CurrentPrintFile);
```
Cancel the file currently being printed.

Declaration
```
procedure CopyEnv(Src, Dest : EnvRec);
```

Purpose
Copy contents of Src environment to Dest environment.

Description
Copies one environment to another. If the Dest environment is smaller than the Src, CopyEnv copies what it can and truncates the last element wherever it must. The copy never includes the pathname information that DOS 3.0+ tacks at the end of the normal environment. If this information is needed, you can usually use the SetProgramStr function to copy it. See ShellWithPrompt (OPDOS.PAS) for an example of how to do this.

Example
```
var CEnv, NEnv : EnvRec;
...
CurrentEnv(CEnv);
NewEnv(NEnv, CEnv.EnvLen);
CopyEnv(CEnv, NEnv);
...
DisposeEnv(NEnv);
```
Creates a copy of the current environment in NEnv.

See Also
NewEnv SetProgramStr

CopyFile / CurrentEnv

Declaration

```
function CopyFile(SrcPath, DestPath : String; Buffer : Pointer;
                  BufferSize : Word) : Word;
```

Purpose

Copy the file specified by SrcPath to DestPath.

Description

This routine makes a copy of the file specified by SrcPath in DestPath. DestPath *must* be a complete filename; it may not be the name of a drive or directory. Buffer must point to a region of memory of no less than BufferSize bytes that has already been allocated.

This is a low level routine, and the input pathnames are not checked for validity. If an error occurs during the copy process, the destination file (DestPath) will be deleted, and an error code will be returned in the function result. Possible function results are:

```
0  Success
1  Unable to open source file
2  Unable to open destination file
3  Error reading from source file
4  Error writing to destination file
5  Error getting date/time stamp for source file
6  Error setting date/time stamp for destination file
7  Error closing destination file
```

One error condition that cannot be caught by CopyFile is that of the case where SrcPath and DestPath refer to the same file. Since copying a file onto itself is pointless at best and dangerous at worst, you should call the SameFile function before calling CopyFile to insure that the names refer to different files.

Example

```
var Buffer : array[1..4096] of Byte;
    FName, NewName : string[79];
...
  NewName := 'A:'+JustFileName(FName);
  if (not SameFile(FName, NewName, Status)) and (Status = 0) then
    Status := CopyFile(FName, NewName, @Buffer, SizeOf(Buffer));
```

Copy file FName to drive A. JustFileName is a function in the OPSTRING unit.

```
Status := CopyFile(FName, 'A:', @Buffer, SizeOf(Buffer));
```

Incorrect! DestPath must be a complete filename.

See Also

SameFile

Declaration

```
procedure CurrentEnv(var Env : EnvRec);
```

Purpose

Return current environment record.

Description

Returns a record for the current program's environment. There isn't much space left to add to this environment, and any changes made to it are lost when the current program ends.

Example

See the example in the entry for CopyEnv.

See Also

MasterEnv NewEnv

Declaration
```
function DefaultDrive : Char;
```
Purpose
Return the default drive as an uppercase letter.

Description
DefaultDrive returns a character in the range 'A'..'Z'. To convert the result to a drive number, you'd say
```
DriveNumber := Ord(DefaultDrive)-64;
```
Example
```
WriteLn('Current drive: ', DefaultDrive);
```
Would display 'Current drive: C' if C: is the current drive.

See Also
NumberOfDrives SelectDrive

Declaration
```
procedure DisposeEnv(var Env : EnvRec);
```
Purpose
Deallocate an environment previously allocated on heap.

Description
Deallocates heap space for an environment previously allocated with NewEnv. Calling DisposeEnv with the master or current environment record as a parameter does nothing.

Example
See the example in the entry for CopyEnv.

See Also
NewEnv

Declaration
```
function DosVersion : Word;
```
Purpose
Return the DOS version number.

Description
The result is in BCD (Binary Coded Decimal) format, with the major version number in the high byte, and the minor version number in the low byte. For example, under DOS 3.1, the result would be $030A ($0A = 10); under DOS 2.11, $020B ($0B = 11); under DOS 3.3, $031E ($1E = 30); and so on.

Note: In Turbo Pascal 5.0, Borland added a function called DosVersion to the DOS unit. Unfortunately, it returns the major version in the low byte and the minor version in the high byte–just the opposite of what this routine returns. You would therefore be well-advised to use a "unit override" when calling either routine (OpDos.DosVersion or Dos.DosVersion).

Example
```
if OpDos.DosVersion >= $0400 then
  WriteLn('Warning: Program not tested with DOS 4.0 or higher');
```
Test for DOS 4.00 or higher.

DumpEnv / EndPrintStatus / EnvFree

Declaration
```
procedure DumpEnv(Env : EnvRec);
```

Purpose
Dump the environment to the screen.

Description
Displays all strings stored in the specified environment on the screen (unless Output has been redirected to a file). The listing is terminated with the number of unused bytes in the environment.

Example
See the example program in the entry for SetEnvStr.

See Also
SetEnvStr

Declaration
```
procedure EndPrintStatus;
```

Purpose
Releases the spooler from the GetPrintStatus procedure.

Description
Reenables the print spooler after a call to GetPrintStatus, so that printing can resume.

See Also
GetPrintStatus

Declaration
```
function EnvFree(Env : EnvRec) : Word;
```

Purpose
Return bytes free in environment.

Description
EnvFree returns the number of bytes that are unused in the specified environment. You may want to check this value before adding a new string to the environment, although SetEnvStr checks as well.

Example
```
S := 'New string';
if EnvFree(Env) >= (Length('NEW='+S)+1) then
  Added := SetEnvStr(Env, 'NEW', S);
```

Adds 'NEW=New string' to the specified environment.

See Also
SetEnvStr

Declaration

```
function ExecDos(Command : string; UseSecond : Boolean;
                 EDP : ExecDosProc) : Integer;
```

Purpose

Execute any DOS command.

Description

ExecDos, which acts as a front end for Turbo Pascal's Exec procedure, compresses the heap temporarily in order to free up as much memory as possible for the program being executed. The heap is restored to its original condition after the Exec call.

Command is the name of a program or DOS command to be executed, or an empty string (which means "shell to DOS "). UseSecond determines whether a second copy of COMMAND.COM is loaded to process the Exec call. If it is False, Command must be the full pathname of the program to be executed. (See the entry for the ExistOnPath function.)

EDP specifies a user-written routine that will be called to display status information, etc. It should be of the form

```
{$F+}
procedure MyExecDosProc(ActionCode : ActionCodeType; Param : Word);
begin
end;
```

It must be a global (non-nested) procedure compiled under the FAR model. If you don't want to write a routine of this type, pass 'NoExecDosProc' (a do-nothing routine declared by OPDOS) as your third parameter. For a discussion of the ExecDos action codes, see the entry for ActionCodeType.

ExecDos return codes are as follows:

```
 0 Success
-1 Insufficient memory to store Turbo's free list
-2 DOS setblock error before EXEC call
-3 DOS setblock error after EXEC call (critical error)
-4 Insufficient memory to run DOS command
else a DOS error code
```

ExecDos calls the SwapVectors routine in the DOS unit to temporarily restore all interrupt vectors grabbed by the SYSTEM unit. It cannot, however, restore vectors taken over by other units--such as OPENHKBD--and it is your responsibility to do so when necessary.

Warning: Never use ExecDos to execute a TSR. ExecDos gives away part of your program's memory allocation on the assumption that it will be able to recover it for you. If it is unable to do so, as it would be if you tried to EXEC a TSR, it will return an error code, usually -3.

Note too that ExecDos cannot be used by a program running "in memory" within Turbo Pascal's integrated environment. If you try, ExecDos will return an error code, usually -2.

Examples

```
Code := ExecDos('C:\TURBO\TPC.EXE '+ProgName, False, NoExecDosProc);
```

Call Turbo to compile ProgName. Since the UseSecond parameter is False, the complete pathname for TPC.EXE must be specified.

```
Code := ExecDos('MAKEMENU', True, NoExecDosProc);
```

Execute MAKEMENU. A second copy of COMMAND.COM will be loaded to find and execute the program.

```
var
  P : Pointer;
  BufSize : Word;
  Code : Integer;

{$F+}
procedure MyExecDosProc(ActionCode : ActionCodeType; Param : Word);
var
  Key : Word;
begin
  case ActionCode of
    ExecSaveScreen :    {don't allocate screen buffer, save screen}
      if not SaveWindow(1, 1, ScreenWidth, ScreenHeight, False, P) then
        {can't fail--buffer already allocated};
    ExecShowMemory :    {show memory available for shell}
      WriteLn('Approximately ', Param, 'K bytes available');
    ExecPauseAfterRun : {make sure screen isn't cleared too fast}
      begin
        Write('Press any key to continue...');
        Key := ReadKeyWord;
      end;
    ExecRestoreScreen :  {restore screen, don't dispose of screen buffer}
      RestoreWindow(1, 1, ScreenWidth, ScreenHeight, False, P);
  end;
end;
{$F-}
...
{allocate a screen buffer}
BufSize := 2*ScreenWidth*ScreenHeight;
GetMem(P, BufSize);
Code := ExecDos('', True, MyExecDosProc);
FreeMem(P, BufSize);
```

Shell to DOS. MyExecDosProc is a user-written action code handler.

See Also

ExecDosSwap (OPEXEC)
ExistOnPath
MinSpaceForDos
ShellWithPrompt
StackSafetyMargin

Declaration

```
function ExistFile(FName : string) : Boolean;
```

Purpose

Return True if FName is found.

Description

ExistFile does not accept wildcard characters, and it looks only in the current directory or in the directory specified in FName.

Examples

```
Found := ExistFile('C:\AUTOEXEC.BAT');
```

ExistFile would look for 'AUTOEXEC.BAT' in 'C:\'.

```
Found := ExistFile('AUTOEXEC.BAT');
```

ExistFile would look in the current directory for 'AUTOEXEC.BAT'.

```
Found := ExistFile('*.DOC');
```

Incorrect! ExistFile does not accept wildcard characters.

Declaration

```
function ExistOnPath(FName : string; var FullName : string) : Boolean;
```

Purpose

Return True and a usable pathname if FName is found.

Description

Unlike ExistFile, this routine can often locate files that are not in the current directory. If FName contains a drive and/or path identifer, only the specified drive and directory is searched. But if only a filename is specified, ExistOnPath will search, in order:

- the current directory
- the executable program's directory (under DOS 3.0 or higher only)
- all directories on DOS's PATH

It is therefore particularly useful for locating any data files, configuration files, overlay files, and so on that might be required by your program. Like ExistFile, ExistOnPath does not accept wildcards.

ExistOnPath will *not* return a complete pathname if FName is found in the current directory. The reason is that the user may have a "path enhancement" utility installed (i.e., a TSR that supplements DOS's PATH searching facility). If such a utility is in use, a statement like

```
ExistFile('MYPROG.DAT');
```

can return True even if MYPROG.DAT is not in the current directory, and there would be no way for ExistOnPath to determine for certain where MYPROG.DAT really is. If you are not concerned about such cases, and you want a complete pathname for a file found by ExistOnPath, you should call the FExpand routine in the DOS unit afterward.

Examples

```
Found := ExistOnPath('C:\AUTOEXEC.BAT', FullName);
```

ExistOnPath would look for 'AUTOEXEC.BAT' only in 'C:\'.

```
Found := ExistOnPath('AUTOEXEC.BAT', FullName);
if Found then FullName := FExpand(FullName);
```

ExistOnPath would do a full search for 'AUTOEXEC.BAT'. If the file is found, FullName will contain a complete pathname.

```
Found := ExistOnPath('*.DOC', FullName);
```

Incorrect! ExistOnPath does not accept wildcard characters.

FileHandlesLeft / FileHandlesOpen

Declaration

```
function FileHandlesLeft : Byte;
```

Purpose

Return the number of available file handles.

Description

This routine simply attempts to open up to 20 files, and returns the number that it could successfully open. All files that were opened are closed on exit. This routine is intended primarily for purposes of debugging.

Please keep in mind that FileHandlesLeft uses nearly 3K of stack space.

Example

```
{$IFDEF Debug}
WriteLn('There are ', FileHandlesLeft, ' file handles available.');
{$ENDIF}
```

A typical use for FileHandlesLeft, a debugging statement that displays the number of file handles currently available.

Declaration

```
function FileHandlesOpen(CountDevices : Boolean) : Byte;
```

Purpose

Return the number of open files owned by a program.

Description

This routine returns the number of files that your program currently has open. The CountDevices parameter indicates whether or not the standard DOS devices (those with handles in the range StdInHandle..StdPrnHandle) should be included in the count. The number returned will always be in the range 0..20 if CountDevices is True, or 0..15 if it is False. Keep in mind that there is no guarantee that all 20 file handles will be available for use by your program unless "FILES=20" (or a number greater than 20) appears in the user's CONFIG.SYS file. The default setting for FILES is only 8, unfortunately. This routine is intended primarily for purposes of debugging.

Example

```
{$IFDEF Debug}
WriteLn('There are now ', FileHandlesOpen(True), ' files open.');
{$ENDIF}
```

A typical use for FileHandlesOpen, a debugging statement that displays the number of files that are currently open.

Declaration

```
function FlushDosBuffers(var F) : Boolean;
```

Purpose

Flush DOS's buffers for the specified file.

Description

This routine forces DOS to flush all buffered data for the specified file. F is declared as an untyped var parameter to allow the same routine to work for both typed ('F : File of Byte') and untyped ('F : File') files. FlushDosBuffers should *not* be used for text files; TextFlush should be used instead. FlushDosBuffers returns False if an I/O error occurs.

Example

```
var F : File;
    Result : Boolean;
...
Result := FlushDosBuffers(F);
```

Correct.

```
var F : Text;
...
Result := FlushDosBuffers(F);
```

Incorrect! FlushDosBuffers should not be used on text files. Use TextFlush.

See Also

TextFlush

Declaration

```
function GetDiskClass(Drive : Char; var SubstDriveChar : Char) : DiskClass;
```

Purpose

Return the disk class for the drive with the specified letter.

Description

Given a drive letter ('A'..'Z'), this routine returns a value indicating what kind of disk is installed in the specified drive. The possible return values are described in the entry for the type DiskClass earlier in this chapter.

There are two special cases that you should be aware of. First, the DOS ASSIGN program allows you to treat one existing drive as another. For example

```
ASSIGN B=C
```

would cause all disk I/O pertaining to drive B to be routed to drive C. Similarly, the DOS SUBST program allows you to create a new logical drive from an existing drive or directory. For example,

```
SUBST H: C:\TURBO
```

would allow you to refer to the directory "C:\TURBO" as drive H.

GetDiskClass is able to recognize both of these situations (although it cannot distinguish between them). When it does, it will return 'SubstDrive' as its function result, and it will pass back the letter of the actual drive being used for the specified Drive in SubstDriveChar. In the case of the second example above (involving SUBST), SubstDriveChar would be set to 'C' if the drive parameter were 'H'. Note that SubstDriveChar will always be equal to Drive if the function result is something other than SubstDrive.

Limitations: GetDiskClass cannot recognize the type of a floppy drive, only the type of the disk that is in it (if any). For example, a 1.2 meg floppy drive will be classified as Floppy360 if it has a 360K disk in it. Also, GetDiskClass does not currently recognize optical drives, nor does it identify several rarely used floppy disk types (single-sided floppies, for example).

GetDiskInfo / GetDta

Declaration

```
function GetDiskInfo(Drive : Byte; var ClustersAvailable, TotalClusters,
                     BytesPerSector, SectorsPerCluster : Word) : Boolean;
```

Purpose

Return technical info about the specified drive.

Description

This routine returns information about the organization of a given drive. You can use it to calculate the total capacity of a disk, the number of bytes free, or to obtain data to be used in conjunction with ReadDiskSectors and/or WriteDiskSectors.

Drive is a number indicating the drive of interest: 0 = default, 1 = A, 2 = B, and so on. To convert a drive letter to a drive number, use the following formula:

```
DriveNumber := Ord(UpCase(DriveLetter))-64;
```

The function returns False if the drive number is invalid. Otherwise, it returns the following values: ClustersAvailable (number of unused clusters on the disk), TotalClusters (total number of clusters on the disk), BytesPerSector (number of bytes in a given disk sector, almost always 512), and SectorsPerCluster (number of sectors in a cluster, usually 1, 2, or 4).

Formulas:

```
BytesPerCluster := LongInt(SectorsPerCluster) * LongInt(BytesPerSector);
DiskCapacity := LongInt(TotalClusters) * BytesPerCluster;
FreeSpace := LongInt(ClustersAvailable) * BytesPerCluster;
```

Example

```
var BytesPerCluster, DiskCapacity, FreeSpace : LongInt;
    ClustersAvailable, TotalClusters,
    BytesPerSector, SectorsPerCluster: Word;
...
if GetDiskInfo(0, ClustersAvailable, TotalClusters, BytesPerSector,
               SectorsPerCluster) then begin
    BytesPerCluster := LongInt(SectorsPerCluster) * BytesPerSector;
    DiskCapacity := LongInt(TotalClusters) * BytesPerCluster;
    FreeSpace := LongInt(ClustersAvailable) * BytesPerCluster;
    WriteLn('Info about drive ', DefaultDrive, ':');
    WriteLn(' Bytes per sector:  ', BytesPerSector);
    WriteLn(' Bytes per cluster: ', BytesPerCluster);
    WriteLn(' Disk capacity:     ', DiskCapacity);
    WriteLn(' Bytes available:   ', FreeSpace);
end;
```

See Also

ReadDiskSectors WriteDiskSectors

Declaration

```
procedure GetDta(var DTAptr : Pointer);
```

Purpose

Return the DOS DTA pointer.

Description

GetDta returns a pointer to the currently active DOS DTA. This routine is needed when the DTA is to be temporarily redirected with SetDta, then restored to the original one.

GetDta is provided for backward compatibility with Turbo Professional 5.0.

See Also

SetDta

Declaration

```
function GetEnvironmentString(SearchString : string) : string;
```

Purpose

Return a string from the current environment.

Description

This routine allows you to locate strings assigned to environment variables , such as 'COMSPEC', 'PATH', and 'PROMPT'. The search is case-insensitive, and you do not need to specify '=' at the end of the search string.

Examples

```
WriteLn(GetEnvironmentString('comspec='));
WriteLn(GetEnvironmentString('comspec'));
WriteLn(GetEnvironmentString('COMSPEC='));
WriteLn(GetEnvironmentString('COMSPEC'));
```

All OK, produces the same result--e.g., 'C:\COMMAND.COM'.

```
WriteLn(GetEnvironmentString('COMS'));
```

Correct only if the complete variable name being sought is 'COMS'. This would not locate 'COMSPEC'.

```
WriteLn(GetEnvironmentString(' COMSPEC '));
```

Incorrect! The spaces at the ends of the string must be eliminated.

Declaration

```
function GetEnvStr(Env : EnvRec; Search : string) : string;
```

Purpose

Return a string from the specified environment.

Description

GetEnvStr returns the value of a particular environment string. For example, to check the value of the COMSPEC environment setting in the master environment, you would code a sequence like the following:

```
var
  E : EnvRec;
begin
  MasterEnv(E);
  WriteLn('Master setting for COMSPEC is ', GetEnvStr(E, 'COMSPEC'));
end.
```

The search string may end with '=' if desired, but GetEnvStr will append an equal sign if needed. Searching is not case sensitive. White space in the search string is included in the search and should generally be avoided. If GetEnvStr finds no matching environment string, it returns an empty string.

Example

See the example program in the entry for SetEnvStr.

See Also

SetEnvStr

GetFileMode

Declaration

```
function GetFileMode(FName : string; var Attr : Word) : Byte;
```

Purpose

Get the attribute of a file.

Description

This routine simply puts a "shell" around Turbo's GetFAttr routine to make it easier to use. Returns FName's attribute in Attr, and the DOS error code (DosError) as the function result. The attribute is bit-coded as follows:

Bit	Val	Meaning
0	$01	Read-only
1	$02	Hidden
2	$04	System
3	$08	Volume label
4	$10	Subdirectory
5	$20	Archive bit

To see if a file is hidden, for example, you would use the following test:

```
Hidden := (Attr and $02) <> 0;
```

Example

```
var
  FName : string;
  Attr : Word;
begin
  Write('Enter file name:');
  ReadLn(FName);
  if GetFileMode(FName, Attr) <> 0 then
    WriteLn('File not found')
  else begin
    WriteLn(FName, ' is a ');
    if (Attr and $10) <> 0 then
      WriteLn('directory')
    else
      WriteLn('file');
  end
end;
```

Declaration

```
function GetPrintStatus(var QPtr : Pointer) : Byte;
```

Purpose

Halts printing, returns current error status, puts pointer to the filename queue in the QPtr variable.

Description

This routine returns a pointer to a series of ASCIIZ strings representing the names of files in the print queue. Each entry is 64 bytes in size, and the end of the queue is marked by a name starting with a null. Printing is suspended from the time you call GetPrintStatus until you issue a call to EndPrintStatus. Note that you can easily convert ASCIIZ strings to normal strings using the Asc2Str routine in OPASCIIZ. GetPrintStatus returns 0 if successful.

Example

```
var
  P : Pointer;
  AP : AsciizPtr absolute P;
  POffset : Word absolute P;
begin
  if GetPrintStatus(P) = 0 then
    while Char(P^) <> #0 do begin {check for null}
      WriteLn(Asc2Str(AP^));      {display current entry}
      Inc(POffset, 64);           {point to next entry}
    end;
  EndPrintStatus;                 {release print spooler}
end;
```

See Also

Asc2Str (OPASCIIZ) EndPrintStatus

Declaration

```
function GetProgramStr(Env : EnvRec) : string;
```

Purpose

Return the name of the program that owns Env, '' if DOS < 3.0 or unknown.

Description

Under DOS 3.0 or higher, this routine returns the name of the program that owns the specified environment.

Example

```
ParentEnv(Env);
S := GetProgramStr(Env);
if S <> '' then
  WriteLn('Parent program is ', S);
```

See Also

SetProgramStr

HandleIsConsole / IsDirectory

Declaration

```
function HandleIsConsole(Handle : Word) : Boolean;
```

Purpose

Return True if handle is the console device.

Description

HandleIsConsole allows you, among other things, to determine whether or not a program's I/O has been redirected. This information can be crucial when writing a filter program, which needs to know if the user has redirected input or output as required. A more common use for the routine, however, is to insure that a user has not asked you to copy data to the 'CON' device. For example, OPEDITOR uses HandleIsConsole before performing a block write operation, since block copying text to 'CON' would make a mess on the screen.

Examples

```
uses Dos;
...
if HandleIsConsole( TextRec(Input).Handle ) then
  WriteLn('Input must be redirected from a file');
```

Test to see if user has redirected standard input.

```
if HandleIsConsole( TextRec(Output).Handle ) then
  WriteLn('Output should be redirected to a file');
```

Test to see if user has redirected standard output.

Declaration

```
function IsDirectory(FName : String) : Boolean;
```

Purpose

Return True if FName is a directory.

Description

This routine may be used to determine whether not a given directory name is valid. Directory names should not end with a '\' except in the case of a root directory. Drive names followed by colons ('A:', 'B:', 'C:') are considered to be valid directories if the drive is valid.

Examples

```
IsDirectory('C:\')
IsDirectory('C:')
```

Both return True if C is a valid drive.

```
IsDirectory('C:\TEST')
```

Returns True if the drive and directory are both valid.

```
IsDirectory('C:\TEST\')
```

Returns False due to the trailing backslash.

See Also

ValidDrive

Declaration

procedure MasterEnv(var Env : EnvRec);

Purpose

Return master environment record.

Description

Returns a record for the master system environment. Changes to this environment are retained after the current program exits. Finding the master environment is a little tricky, and if for some reason the method doesn't work on your system, the EnvSeg field of the returned environment record will be set to zero. The master environment for the system is typically larger than that for the current program--DOS copies only the portion of the master environment that is actually in use when it runs each program.

Example

```
var Env : EnvRec;
...
MasterEnv(Env);
if Env.EnvSeg <> 0 then
  DumpEnv(Env);
```

Dump the contents of the master environment to the screen.

See Also

CurrentEnv SetEnvStr

Declaration

```
procedure NewEnv(var Env : EnvRec; Size : Word);
```

Purpose

Allocate a new environment on the heap.

Description

The most common use for this routine is in a program that spawns other programs and needs to pass a lot of environment information to them. Size specifies the number of bytes to reserve for the new environment. NewEnv actually allocates 31 bytes more than you request, to assure that paragraph alignment is possible and to create a fake DOS memory control block which keeps other programs and routines happy. If there isn't sufficient heap space to allocate the new environment, NewEnv returns the EnvSeg field of Env set to zero.

After a call to this routine, the new environment is totally filled with nulls (#0). Use the CopyEnv routine to copy from another environment into the new one.

Example

See the example for CopyEnv.

See Also

CopyEnv DisposeEnv
CurrentEnv SetCurrentEnv

NumberOfDrives / OpenStdDev / ParentEnv

Declaration
```
function NumberOfDrives : Byte;
```

Purpose
Return the number of logical drives.

Description
Under DOS 3.0 or greater, the value returned will always be at least 5, because these versions support SUBSTituted drives. Under DOS 2.x, the actual number of drives will be returned.

Example
```
WriteLn('Number of drives according to DOS: ', NumberOfDrives);
```

Declaration
```
function OpenStdDev(var F : Text; StdHandle : Word) : Boolean;
```

Purpose
Assign the text file to a standard DOS device: 0, 1, 2, or 4.

Description
This routine lets you easily access the standard input, standard output, standard error output, and standard printer devices. F is any unopened text file variable, and StdHandle is 0 (StdInHandle), 1 (StdOutHandle), 2 (StdErrHandle), or 4 (StdPrnHandle). Among other things, OpenStdDev allows you to write to the standard output device in programs that use CRT or OPCRT. It returns False only if StdHandle is an invalid number.

Example
```
var StdOut : Text;
...
if not OpenStdDev(StdOut, StdOutHandle) then {error};
```

Declaration
```
procedure ParentEnv(var Env : EnvRec);
```

Purpose
Return environment record of program's parent.

Description
This routine returns the environment record for the program that spawned it (its parent). If the current program was executed from the DOS command line, the parent environment will be DOS's master environment. Otherwise the environment will be that of the program that performed the DOS EXEC call.

Example
See the example for SetEnvStr.

See Also
MasterEnv

Declaration

```
function ParsePath(var InputPath, SearchPath, LeadInPath : string) : Boolean;
```

Purpose

Takes a user entered path, trims blanks, and returns a valid global search path and a valid lead-in path.

Description

This routine complements the FindFirst and FindNext routines in the standard DOS unit. ParsePath parses a file mask (InputPath) in the same manner that the DOS COPY command does, and returns a string to be passed on to FindFirst (SearchPath) and another string containing just the path being searched. Leading and trailing blanks are removed, and all characters are converted to upper case. For example, given the string ' \dos ', ParsePath would return '\DOS*.*' in SearchPath and '\DOS\' in LeadInPath. It returns False if InputPath cannot be parsed.

Note that ParsePath's validation is incomplete, and in some cases it will return True even if the InputPath specifies an invalid drive (for example).

Examples

Given an InputPath of 'c:' ParsePath would return

```
'C:*.*' {SearchPath}
'C:'    {LeadInPath}
```

Given an InputPath of '*.COM' ParsePath would return

```
'*.COM' {SearchPath}
''      {LeadInPath}
```

Declaration

```
function PrintInstalled : Boolean;
```

Purpose

Return True if PRINT.COM is installed.

Description

All of the routines for accessing DOS's print spooler (PRINT.COM) require DOS 3.0 or higher. If the program is being run under an earlier version, PrintInstalled will always return False. Otherwise, it will return False only if the print spooler is not resident.

Example

```
if not PrintInstalled then
  WriteLn('PRINT.COM not resident');
```

See Also

CancelAllPrintFiles
CancelPrintFile
EndPrintStatus
GetPrintStatus
SubmitPrintFile

ReadDiskSectors

Declaration

```
function ReadDiskSectors(Drive : Word; FirstSect : Longint;
                         NumSects : Word; var Buf) : Boolean;
```

Purpose

Read absolute disk sectors.

Description

This routine provides you access to individual disk sectors , allowing you to inspect even unused portions of a disk. Drive is a number between 0 (= A:) to 25 (= Z:). To convert a drive letter to a drive number, use the following formula:

```
DriveNumber := Ord(UpCase(DriveLetter))-Ord('A');
```

FirstSect is the sector to read from (the first sector on a disk is 0). This value should normally be in the range 0..65535, but if DOS 3.31 or higher is in use the number may be larger. If DOS 3.3 or lower is in use, the high word of FirstSect will be ignored.

NumSects is the number of sectors to read. Buf is the buffer to read data into; it must be at least NumSects * BytesPerSector bytes (see GetDiskInfo). If ReadDiskSectors fails, it returns False and puts an error code in DosError, interpreted as follows:

```
High byte (error code generated by BIOS):
```

128	Time out: drive failed to respond
64	Seek operation failed
32	Controller failed
16	Bad CRC (Cyclical Redundancy Check) -- parity error
8	DMA (Direct Memory Access) failure
4	Sector not found
3	Attempt to write on write protected diskette
2	Bad address mark: sector ID marking invalid or not found

```
Low byte (error code generated by DOS):
```

12	General failure
11	Read error
10	Write error
8	Sector not found
7	Unknown media; disk format not recognized
6	Seek error
4	Bad CRC
2	Drive not ready
1	Unknown unit: invalid drive number
0	Attempt to write on write protected diskette

Example

```
var Buffer : array[1..2048] of Byte;
    CA, TC, BPS, SPC : Word;
...
if not GetDiskInfo(0, CA, TC, BPS, SPC) then begin
  WriteLn('Error reading from drive ', DefaultDrive); Exit;
end;
if not ReadDiskSectors(Ord(DefaultDrive)-Ord('A'), 0, 1, Buffer) then begin
  WriteLn('Error ', DosError, ' reading from drive ', DefaultDrive);
  DosError := 0;
  Exit;
end;
```

See Also

GetDiskInfo WriteDiskSectors

Declaration

```
function SameFile(FilePath1, FilePath2 : String; var ErrorCode : Word) : Boolean;
```

Purpose

Return True if FilePath1 and FilePath2 refer to the same physical file.

Description

This routine is most typically used in conjunction with the CopyFile function. It tests to see if two filenames (FilePath1 and FilePath2) refer to the same physical file. FilePath1 is assumed to refer to an existing file. FilePath2 need not name an existing file, but its path, if any, should be valid.

Because a simple comparison of the two filenames would not yield a definitive answer, SameFile accomplishes its task by trying to obtain the file attributes for both files. If this operation succeeds, SameFile then changes the attributes for FilePath1 temporarily and rereads the attributes for FilePath2. If the attributes for FilePath2 have changed, then SameFile returns True. Otherwise it returns False.

Possible error codes:

```
0  Success (files are different or FilePath2 does not exist)
1  FilePath1 does not exist or could not be accessed
2  FilePath2 has invalid path or could not be accessed
3  Error getting/setting file attributes
```

Example

See the example for CopyFile.

See Also

CopyFile

Declaration

```
procedure SelectDrive(Drive : Char);
```

Purpose

Selects the specified drive as default if possible.

Description

Drive should be a letter in the range 'a'..'z' or 'A'..'Z'. If the drive is invalid, SelectDrive does nothing.

Examples

```
SelectDrive('A');
SelectDrive('b');
SelectDrive('C');
```

See Also

DefaultDrive NumberOfDrives

SetBlock / SetCurrentEnv / SetDta

Declaration

```
function SetBlock(var Paragraphs : Word) : Boolean;
```

Purpose

Change size of DOS memory block allocated to this program.

Description

This routine, intended primarily for internal use by ExecDos, allows you to increase or decrease the amount of memory allocated to your program. Paragraphs is the new size of the memory block requested. If the request cannot be filled, SetBlock returns False and Paragraphs contains the number of available paragraphs.

Very important: If you change a program's memory block, you must also be certain that Turbo's free list is set up properly. The ExecDos routine in OPDOS.PAS demonstrates how to safely alter the memory block.

Examples

See the ExecDos routine in OPDOS.PAS for examples.

See Also

ExecDos

Declaration

```
procedure SetCurrentEnv(Env : EnvRec);
```

Purpose

Specify a different environment for the current program.

Description

Sets the environment of the current program to a different one, the one specified by the Env parameter. You'll want to do this after calling NewEnv and initializing the new environment, but before spawning a child process. DOS copies the current environment to the child process. To keep from messing up DOS's own memory allocation, you should always restore the current environment to the one DOS allocated before quitting a program.

Example

See the calls to SetCurrentEnv in ShellWithPrompt (OPDOS.PAS).

See Also

NewEnv ShellWithPrompt

Declaration

```
procedure SetDta(DTAptr : Pointer);
```

Purpose

Set the DOS DTA to point to DTAptr.

Description

DOS uses a "Disk Transfer Area" for a number of disk operations, notably for directory scanning. A common technique is to point the DOS DTA to a region overlayed by a Pascal record. After DOS writes directory information to this region, the fields of the Pascal record provide an easy means of parsing the data. In fact, this is the technique used by the FindFirst and FindNext routines in the DOS unit, and because FindFirst and FindNext reset the DTA for you automatically, you'll probably never need to use SetDta or its companion routine GetDta.

SetDta is provided for backward compatibility with Turbo Professional 5.0.

See Also

GetDta

Declaration

```
function SetEnvStr(Env : EnvRec; Search, Value : string) : Boolean;
```

Purpose

Set environment string, returning true if successful.

Description

Sets the value of an environment string. Note that you can set the value in the master or current environment, depending on how the Env parameter was initialized. Search specifies the name of the environment string to change, for example, 'COMSPEC'. Value specifies its new value. If Value is an empty string, the entire item is removed from the environment. New values are always added to the end of the environment, after first removing any replaced value from the middle. If there is insufficient space for the new environment value, SetEnvStr returns False, otherwise true.

This routine has one side effect. If the environment to be changed is the current environment, SetEnvStr wipes out the program path name stored beyond the end of the environment under DOS 3.0 and later. If you need to access this pathname, be sure to do so (using 'ParamStr(0)' or GetProgramStr) before calling SetEnvStr (see the example for SetEnvStr).

Example

```
uses
  opstring, opdos, opcrt, opcmd, opsedit;
var SLE : SimpleLineEditor;
    E, S : string;
    Env : EnvRec;
begin
  E := StUpcase(ParamStr(1));
  if E = '' then begin
    WriteLn('Usage: EDITENV EnvironmentVariableName');
    WriteLn('       To edit EnvironmentVariableName');
    WriteLn('   Or: EDITENV /D');
    WriteLn('       To dump the contents of the environment');
    Halt;
  end;
  ParentEnv(Env);
  if (E = '/D') or (E = '-D') then begin
    DumpEnv(Env); Halt;
  end;
  S := GetEnvStr(Env, E);
  if S = '' then begin
    WriteLn('"', E, '" not found in environment'); Halt(1);
  end;
  SLE.Init(DefaultColorSet);
  WriteLn('Current value: ', S);
  SLE.ReadString('New value:     ', WhereY, 1, 127, ScreenWidth-16, S);
  if (SLE.GetLastCommand <> ccQuit) then begin
    WriteLn;
    if SetEnvStr(Env, E, S) then
      WriteLn('Environment string changed')
    else
      WriteLn('Unable to change environment string');
  end;
end.
```

A simple program that lets you edit a string in DOS's master copy of the environment (or the parent program's copy, if you are running within a DOS shell) or to display its contents.

SetProgramStr / SetRawMode

Declaration

```
function SetProgramStr(Env : EnvRec; Path : string) : Boolean;
```

Purpose

Add a program name to the end of an environment if sufficient space.

Description

Provides a method to set the program path name, normally available under DOS 3.0 or later. Path is appended to the specified environment just as DOS would do it. If insufficient space remains, SetProgramStr does nothing and returns False.

Note that this routine should never be used on DOS's master environment, only on the current program's copy of the environment or a special environment created with NewEnv.

Example

```
S := GetProgramStr(Env);
SetEnvStr(Env, 'NEW', 'New string');
if not SetProgramStr(Env, S) then {insufficient space};
```

Shows how to save and restore the program name while changing an environment string with SetEnvStr.

See Also

GetProgramStr
SetEnvStr
NewEnv

Declaration

```
procedure SetRawMode(var F : Text; On : Boolean);
```

Purpose

Set "raw" mode on or off for the specified text file (must be a device).

Description

This routine allows you to change the mode of a character device (standard input, standard output, standard printer) from "cooked" to "raw" and back again. In cooked mode, DOS expands tabs and interprets control characters such as ^S and ^C when reading from or writing to a character device. In raw mode, all characters are treated as data.

The F parameter passed to SetRawMode must be an open text file associated with a character device. If F is not a device, it does nothing. I/O errors are reported in Dos.DosError.

Probably the most common use for this routine is to put the standard printer device into raw mode. Another common use for SetRawMode is to speed up I/O operations involving the standard input or standard output devices.

Example

```
var
  F : Text;
begin
  Assign(F, 'LPT1');
  Rewrite(F);
  SetRawMode(F, True);
  ...
  Close(F);
  SetRawMode(F, False);
end.
```

Put the printer device in raw mode at beginning of program, and restore cooked mode at the end. In most cases, an exit handler should also be installed that will restore cooked mode even if the program ends prematurely.

Declaration

```
function ShellWithPrompt(Prompt : string; EDP : ExecDosProc) : Integer;
```

Purpose

Shell to DOS with a new prompt.

Description

Starts a DOS shell with the prompt set to the specified value. The user can execute DOS commands until EXIT is entered, and then the calling program continues. This routine depends on the ExecDos routine for the mechanics of the shell.

The Prompt string may specify any valid DOS prompt, including the metacommands such as pg, and ANSI escape sequences if ANSI is installed. The prompt totally replaces the existing prompt until the shell returns.

ShellWithPrompt allocates heap space for a new environment including the specified Prompt. Heap space is allocated even if the prompt would fit in the existing environment. The new environment is a copy of the current one, with the PROMPT entry changed.

ShellWithPrompt returns a status value in the function result. The value will be zero for a successful exec. Positive values are interpreted as DOS error codes. Negative values are error codes specific to ExecDos and ShellWithPrompt (codes -1..-4 are exactly the same as ExecDos, ShellWithPrompt adds -5..-7):

```
 0   Success
-1   Insufficient memory to store Turbo's free list
-2   DOS setblock error before EXEC call
-3   DOS setblock error after EXEC call (critical error)
-4   Insufficient memory to run DOS command
-5   Can't read current environment
-6   Insufficient heap space for new environment
-7   Space calculation error (shouldn't happen)
else a DOS error code
```

Example

```
NewP := 'Enter "EXIT" to return to MYPROG'^M^J+GetEnvironmentString('PROMPT');
Status := ShellWithPrompt(NewP, NoExecDosProc);
```

Shell to DOS and display 'Enter "EXIT" to return to MYPROG' above the regular DOS prompt.

See Also

ExecDos

SubmitPrintFile / TextFileSize

Declaration

```
function SubmitPrintFile(FileName : string) : Byte;
```

Purpose

This procedure submits a file to PRINT.COM.

Description

This routine lets you submit files for concurrent printing under DOS 3.0 and higher, provided that PRINT.COM is installed. Before calling it, you should be certain that both of these conditions are met by calling PrintInstalled.

If the print request can be filled, SubmitPrintFile returns 0; otherwise, one of the following error codes:

```
 1  Invalid function
 2  File not found
 3  Path not found
 4  Too many files open
 5  Access denied
 8  Print queue full
 9  Busy
12  Name too long
15  Invalid drive
```

Example

```
if not PrintInstalled then
  WriteLn('PRINT.COM not installed')
else if SubmitPrintFile('\AUTOEXEC.BAT') <> 0 then
  WriteLn('Cannot print file');
```

See Also

PrintInstalled

Declaration

```
function TextFileSize(var F : Text) : LongInt;
```

Purpose

Return the size of text file F.

Description

This routine allows you to determine the current size of any text file opened for input or output. The current contents of the text buffer are accounted for in the result. TextFileSize returns -1 in case of I/O error.

Example

```
var MyFileSize : LongInt;
  F : Text;
begin
  {--open F--}
  MyFileSize := TextFileSize(F);
  if MyFileSize = -1 then WriteLn('Error')
  else WriteLn('Size = ', MyFileSize);
end;
```

Declaration
```
function TextFlush(var F : Text) : Boolean;
```

Purpose
Flush the buffer(s) for a text file.

Description
This routine empties a text file's buffer. If the file has been opened for output, the current contents of the buffer are written to disk, and DOS's file buffers are flushed using DOS's Duplicate File Handle ($45) and Close File ($3E) functions. If it has been opened for input, the DOS file pointer is moved back to point to the next character to be read (the location it would be at if there were no buffering). TextFlush returns False in case of I/O error.

Example
```
var F : Text;
...
if not TextFlush(F) then WriteLn('Error flushing F')
else WriteLn('F flushed');
```

Declaration
```
function TextPos(var F : Text) : LongInt;
```

Purpose
Return the current position of the logical file pointer.

Description
TextPos obtains the current position of the physical file pointer, then adjusts that value to account for the current contents of the text file's buffer. The result is the current position of the *logical file pointer*–the position that the physical file pointer would be at if there were no buffering. TextPos returns -1 in case of I/O error. See the entry for TextSeek, below, for an example of how to use TextPos.

Declaration
```
function TextSeek(var F : Text; Target : LongInt) : Boolean;
```

Purpose
Do a Seek for a text file opened for input.

Description
This routine allows you to perform random access on text files opened for input. TextSeek returns False if F is not opened for input or if an I/O error occurs. Note that TextSeek sometimes has to empty the text file's buffer, so for performance reasons it is unwise to use it on files with large buffers created with SetTextBuf. TextSeek will, however, avoid doing so whenever possible: if the data being sought is already in the buffer, it will simply reposition the file's buffer pointer, rather than performing a a Seek into the physical file.

Note that Target specifies a byte offset into F, *not* a line number.

Example
```
var SeekDest : LongInt; F : Text;
begin
  {--open F--}
  Write('Enter seek destination: '); ReadLn(SeekDest);
  if not TextSeek(F, SeekDest) then
    WriteLn('Error')
  else
    WriteLn('SeekDest: ', SeekDest, ' TextPos: ', TextPos(F));
end;
```

TimeMs / ValidDrive

Declaration

```
function TimeMs : LongInt;
```

Purpose

Return time of day in milliseconds since midnight.

Description

This routine provides a convenient means of measuring a program's run time (in thousandths of a second).

Please note that, although the value returned is in milliseconds, the accuracy of the result is limited by the system timer to about 55 milliseconds (1/18th of a second). If you need to time events more accurately, you should use the TPTIMER unit, found on the BONUS disk.

Example

```
var
  Start, Stop : LongInt;
begin
  Start := TimeMs;
  {-- main program code --}
  Stop := TimeMs;
  WriteLn('Elapsed time: ', (Stop-Start)/ 1000.0);
end;
```

Declaration

```
function ValidDrive(Drive : Char) : Boolean;
```

Purpose

Return True if the specified drive is valid.

Description

ValidDrives determines if the specified drive ('A'..'Z') by trying to switch to it temporarily using SelectDrive.

Example

```
for Ch := 'A' to 'Z' do
  if ValidDrive(Ch) then
    WriteLn(Ch, ' is a valid drive');
```

See Also

SelectDrive

Declaration

```
function WriteDiskSectors(Drive : Word; FirstSect : Longint;
                          NumSects : Word; var Buf) : Boolean;
```

Purpose

Write absolute disk sectors. *Use this routine with extreme caution!*

Description

This routine is the counterpart of ReadDiskSectors, and will almost always be used in conjunction with it. When used together, they allow you to read the contents of individual disk sectors into a buffer, alter the contents of the buffer, then write the changes out to disk.

Drive is a number between 0 (= A:) to 25 (= Z:). To convert a drive letter to a drive number, use the following formula:

```
DriveNumber := Ord(UpCase(DriveLetter))-Ord('A');
```

FirstSect is the sector to write to (the first sector on a disk is 0). This value should normally be in the range 0..65535, but if DOS 3.31 or higher is in use the number may be larger. If DOS 3.3 or lower is in use, the high word of FirstSect will be ignored.

NumSects is the number of sectors to write. Buf is the buffer containing the data to be written.

If WriteDiskSectors succeeds, it returns True. Otherwise it returns False and loads DosError (a global variable in the DOS unit) with an error code. See the documentation for ReadDiskSectors for a list of the possible error codes.

Example

```
var
  Drive : Byte;
  Buffer : array[1..2048] of Byte;
  CA, TC, BPS, SPC : Word;
begin
  if not GetDiskInfo(0, CA, TC, BPS, SPC) then begin
    WriteLn('Error reading from drive ', DefaultDrive);
    Exit;
  end;
  Drive := Ord(DefaultDrive)-Ord('A');
  if not ReadDiskSectors(Drive, 0, 1, Buffer) then begin
    WriteLn('Error ', DosError, ' reading from drive ', DefaultDrive);
    DosError := 0; Exit;
  end;
  {--edit contents of Buffer--}
  if not WriteDiskSectors(Drive, 0, 1, Buffer) then begin
    WriteLn('Error ', DosError, ' writing to drive ', DefaultDrive, '!');
    DosError := 0; Exit;
  end;
end;
```

See Also

GetDiskInfo ReadDiskSectors

C: OPEXEC: Swapping Exec Manager

OPEXEC provides a way to squeeze more code into the 640K DOS memory limit. ExecDosSwap, the main procedure in OPEXEC, executes a specified program using the DOS Exec function. Its major advantage over other Exec implementations is that it swaps most of the calling program into expanded memory or to disk and gives all that free memory to the called program. When the subprocess is complete, the calling program is swapped back into place and continues normally.

This technique is especially valuable for menuing environments which must execute other large programs, or modern programming editors which are expected to spawn huge compilations at the touch of a key. In fact, it's useful for any program that must invoke another.

OPEXEC traces its heritage from two earlier units. We wrote our first swapping exec manager for the Turbo Analyst 4.0 integrated environment–which executes the Turbo Pascal command line compiler–so that it could compile larger programs. That unit was provided with the Analyst source code, but it was never documented for general use. The capability was so popular that we wrote a public domain version of that unit along with an article which was published in *Dr. Dobbs Journal*. This public domain unit, EXECSWAP, was a somewhat simplified version of the original one from Analyst. Although it was documented, it didn't include all the bells and whistles expected of a commercial toolbox. OPEXEC combines the best of the Analyst unit and EXECSWAP, and adds some new features inspired by **Object Professional**'s OPDOS and OPSWAP units.

The OPEXEC procedure ExecDosSwap works much like its non-swapping counterpart, ExecDos in the OPDOS unit. Like ExecDos, ExecDosSwap relocates the Turbo Pascal free list, restores SYSTEM interrupt vectors, finds the DOS command processor if needed, and calls the DOC Exec function. In addition, ExecDosSwap allocates EMS or opens a disk file for swapping, and copies the calling program out of memory before the exec and back into memory after the exec. OPEXEC also interfaces some lower level procedures that give you more control over the details of these events.

There are several important guidelines for using OPEXEC:

- list OPEXEC *last* in the USES statement of the main program *only*
- move as much code as possible out of the main program "unit"
- disable all interrupt handling functions of the program before performing the exec call
- set the program's maximum heap limit to 655360

The first two rules are closely related. By following them, you minimize the memory left assigned to your program while the child runs. Taking these rules to the absolute limit results in the following program:

```
program SwapTest;
uses
  Dos, OpExec;
begin
  if ExecDosSwap('', True, nil, 'JUNK.SWP') <> 0 then
    WriteLn('error');
end.
```

Run this program and you'll see that SwapTest keeps only 4K bytes of RAM. The rest of memory is given to COMMAND.COM, which allocates 3-6K bytes for itself and then makes the remainder available to whatever DOS command you enter next.

If you put OPEXEC earlier in the USES statement or put more code in the main program, your program will retain more memory during the exec call. The logic behind this is the same as that for swappable TSR's--see Chapter 11 for more information. One quick tip: if you find that your calling program is retaining more memory than you expect it to do, have the compiler generate a MAP file (/GS option for TPC, Options/Linker/Map File/Segments for TURBO) and see what segments are linked prior to OPEXEC.

When you try putting OPEXEC last in the USES statement of just the main program, you'll quickly notice that there's a catch: you can call OPEXEC routines only from the main "unit". And that conflicts with the recommendation to keep the main unit small. You have two ways to resolve this conflict. The first is to add the calls to the main program and accept the resulting increase in exec overhead. This approach may be quite adequate for smaller applications.

The second is to use a procedure variable to make ExecDosSwap accessible from other units. There is no space overhead for this approach. Here's an example:

```
unit ExecAccess;
interface
uses Dos;
var
  ExecDosSwap : function (Command : string; UseSecond : Boolean;
                          EDP : Pointer; SwapFName : PathStr) : Integer;
implementation
end.

program MainProg;
uses
  Dos,
  ExecAccess,         {Keep this earlier than units that need to use it}
  ...,                {Any other units used by the program go here}
  OpExec;             {Keep this last}
begin
  ExecAccess.ExecDosSwap := OpExec.ExecDosSwap;
  ...
end.
```

ExecAccess is a tiny unit that you create. It's complete as shown here. You're set to go as soon as the main program makes the one assignment statement shown to initialize the procedure variable. Wherever you need to call ExecDosSwap just USE the ExecAccess unit. You'll actually be calling ExecDosSwap indirectly via the procedure variable of the same name stored in ExecAccess, but that doesn't matter, the effect is just the same.

OPEXEC interfaces some typed constants (UseEmsIfAvailable and NoSwapMsgForEms) that you may want to modify at runtime. If so, put the appropriate assignments immediately after the procedure variable initialization shown in the main program.

The first two rules affect the efficiency of OPEXEC. The third rule determines whether your machine will crash or not. Remember that while the child process is running, your application is no longer in memory; it has been overwritten by the child program. As a result, you can't use some common tricks that are possible with a normal Exec call, such as: installing interrupt handlers in the parent that affect the child, sharing data between parent and child (except via the command line), and calling routines in the parent from the child.

ExecDosSwap calls Turbo Pascal's SwapVectors routine, which disables the standard SYSTEM interrupt handlers for you. It is *your* responsibility to disable any other interrupt handlers. Note that **Object Professional's** OPMACRO, OPENHKBD, OPREPLAY, OPTSR, and OPSWAP

units take over interrupt vectors that must be restored. The OPMOUSE unit also installs a mouse event handler if you call EnableEventHanding either directly or indirectly; it is critical that you disable this as well. See each of these units for more information about disabling them.

The final rule--setting the program's maximum heap limit to 655360--is consistent with the recommendation for ExecDos. ExecDosSwap automatically shrinks the Turbo Pascal heap to just the amount you've actually used at the time of the exec. If you reduce the maximum heap you will *restrict* the amount of space available to the child program. Note that this rule is exactly opposite to the one for Borland's Exec procedure in the DOS unit.

OPEXEC offers additional advanced features: a message that appears while the parent swaps itself out of and back into memory; a user-defined procedure that it will call at well-defined times during the exec; and control over EMS usage.

The swap message will keep your users comfortable during the brief delay while the calling application swaps out and the child program loads up. By default, the swap message will appear only if OPEXEC is swapping to or from a disk file. You can disable the message altogether (see procedure SetSwapMsgOn) or force it to appear for both EMS and disk swapping (see typed constant NoSwapMsgForEms). The text of the messages ('Swapping out...' and 'Swapping in...') is hard-coded into OPEXEC.ASM. To change it, you can edit the source file and reassemble.

When you call ExecDosSwap, it goes through several well-defined phases: allocating storage space for the swap, compressing the Turbo Pascal heap and modifying DOS memory allocation, running the child process, and restoring the heap. It's sometimes useful for your program to gain control between these phases, so that you can display status messages, give the user a chance to read the results of the child process, and so on. The user-defined procedure hook of ExecDosSwap gives you this option. This hook is identical to the one provided for the ExecDos function in OPDOS.

We'll provide further details about these options in the discussion that follows.

Constants

```
ExecSaveScreen    = 0;
ExecShowMemory    = 1;
ExecPauseAfterRun = 2;
ExecRestoreScreen = 3;
```

Action codes passed by ExecDosSwap to a user-defined routine. These codes are interpreted just like the ones of the same names in the OPDOS unit. See ExecDosSwap for an example of using these codes.

```
UseEmsIfAvailable : Boolean = True;
```

If this constant is True when PrepareExecWithSwap or ExecDosSwap is called, OPEXEC will attempt to allocate the space needed to store the parent's memory image from expanded (EMS) memory. Using EMS for swapping is preferred over using a disk file: it's faster and there are no nagging fears that the user will remove the disk or erase the swap file. The only reason to set UseEmsIfAvailable to False would be if the child program itself needs a certain amount of EMS memory. If UseEmsIfAvailable is True and enough EMS is available for the swap, OPEXEC will set the EmsAllocated variable to True.

```
NoSwapMsgForEms   : Boolean = True;
```

If this constant is True, OPEXEC will never display a message while swapping to and from EMS. The swap message is enabled or disabled globally by calling the SetSwapMsgOn procedure. For reasonably small applications, it's sensible to display the message only while swapping to disk.

Types

```
ActionCodeType = ExecSaveScreen..ExecRestoreScreen;
```

An ExecDosSwap action code. See the OPDOS descriptions of ExecSaveScreen, ExecShowMemory, ExecPauseAfterRun, and ExecRestoreScreen, and the example for ExecDosSwap, below.

Variables

```
NoOfBytesSwapped : LongInt;
```

After PrepareExecWithSwap has been called, this variable will hold the number of bytes that will be swapped to and from EMS or disk. A user-defined action routine used with ExecDosSwap can also refer to this variable when it is passed the action code ExecShowMemory.

```
EmsAllocated : Boolean;          {True when EMS allocated for swap}
SwapFileAllocated : Boolean;     {True when file allocated for swap}
```

After a successful call to PrepareExecWithSwap, one or the other of these variables will be set True, indicating the location of the swap storage. OPEXEC's exit handler and the RemoveExecWithSwap procedure refer to these variables to decide how to clean up. If you call ExecDosSwap instead of the lower-level routines in OPEXEC, the only opportunity you'll have to see a True value is within a user-defined action routine, since ExecDosSwap deallocates EMS or erases the swap file before it returns.

ExecDosSwap

Declaration

```
function ExecDosSwap(Command : string; UseSecond : Boolean;
                     EDP : Pointer; SwapFName : PathStr) : Integer;
```

Purpose

Execute any DOS command after swapping the parent out of memory.

Description

ExecDosSwap works very much like the ExecDos function in OPDOS. Like ExecDos, it compresses the heap temporarily and restores it after the actual Exec call. The purpose of this heap compression is different for ExecDosSwap, however. For ExecDos, the compression serves to free up memory for the program being executed; for ExecDosSwap, the compression reduces the size of the file or the EMS memory required for the swap. The memory available to the program being executed is controlled exclusively by the order in which the program's units are linked (see this section's introduction for guidelines).

Command is the name of a program or DOS command to be executed, or an empty string (which means "shell to DOS"). UseSecond determines whether a second copy of COMMAND.COM is loaded to process the Exec call. If it is False, Command must be the full pathname of the program to be executed, including its extension.

EDP is the address of a user-written routine that will be called to display status information, etc. The user routine must be of the form

```
{$F+}
procedure MyExecDosProc(ActionCode : ActionCodeType; Param : Word);
begin
end
```

and it must be a global procedure compiled under the FAR model (by using the {$F+} compiler directive). If you don't want to write a routine of this type, pass nil as your third parameter. For a discussion of the ExecDos action codes, see the entry for ActionCodeType in the OPDOS unit, as well as the example below.

SwapFName is the complete pathname of a file to hold the swapped-out parent program. This file must be located on a non-removable drive (not on a floppy) or ExecDosSwap will fail. (Note, however, that ExecDosSwap does not fail for all types of removable media; it classifies a Bernoulli drive as non-removable, for example.) The drive must have sufficient free space to hold the swap file or ExecDosSwap will fail. An existing file of the same name is overwritten without warning. The file is created with the hidden and system attributes so that it won't show up with normal directory searches and can't be deleted inadvertently. When ExecDosSwap returns, it erases the swap file. If EMS is used for the swap, SwapFName is not used.

ExecDosSwap saves a region of memory that extends from a location in the middle of the OPEXEC unit (at the label FirstToSave in OPEXEC.ASM, if you care to look) to the heap's high water mark, HeapPtr, after the free list is relocated into the heap. It initializes NoOfBytesSwapped with the actual number of bytes swapped.

ExecDosSwap return codes are as follows:

```
 0 : Success
-1 : Insufficient memory to store free list
-5 : Unable to allocate EMS or open swap file
else a DOS error code
```

ExecDosSwap calls the SwapVectors routine in the DOS unit to temporarily restore all interrupt vectors grabbed by the SYSTEM unit. It cannot, however, restore vectors taken over by other units--such as OPENHKBD--and it is your responsibility to do so when necessary.

ExecDosSwap won't work with some versions of the popular public domain unit EXTEND, which increases the number of file handles available to a program. The most recent version of EXTEND removes the incompatibility. (This version is referred to as "4" under a new numbering system based on the TLIB librarian. Versions numbered 3.2 or earlier will not

work with ExecDosSwap.) If you don't have access to version 4 or later, be sure to "unextend" before calling ExecDosSwap.

ExecDosSwap leaves the swap file open during the time the child process runs so that it doesn't have to reserve space for the swap file's pathname in the part of OPEXEC that remains resident. Because this file is left open, it's not a good idea to execute a program that makes low-level modifications to the DOS file system. Programs to avoid include CHKDSK, disk defragmenters, and physical directory sorters (like Norton DIRSORT).

Warning: Never use ExecDosSwap to execute a TSR. ExecDosSwap gives away most of your program's memory allocation on the assumption that it will be able to recover it for you. If it is unable to do so, ExecDosSwap will halt your program unceremoniously and return to the DOS command line when it attempts to swap back into memory.

Note too that ExecDosSwap cannot be used by a program running "in memory" within Turbo Pascal's integrated environment. If you try, ExecDosSwap will return an error code, usually 1.

Examples

```
Code := ExecDosSwap('D:\OPRO\MAKEMENU.EXE', False, nil, 'JUNK.SWP');
```

Executes MAKEMENU without the aid of the DOS command processor. The swap file, if used, will be located in the current directory.

```
program SwapTest;
uses Dos, OpCrt, OpExec;
var
  P : Pointer;
  BufSize, Code : Integer;

  {$F+}
  procedure MyExecDosProc(ActionCode : ActionCodeType; Param : Word);
  begin
    case ActionCode of
      ExecSaveScreen :    {don't allocate screen buffer, save screen}
        if not SaveWindow(1, 1, ScreenWidth, ScreenHeight, False, P) then
          {can't fail--buffer already allocated};
      ExecShowMemory :
        begin
          if EmsAllocated then
            WriteLn('Swapping to EMS...')
          else
            WriteLn('Swapping to disk...');
          WriteLn(NoOfBytesSwapped, ' bytes are being swapped');
          WriteLn('Approximately ', Param, 'K of RAM available');
        end;
      ExecPauseAfterRun :
        begin
          Write('Press <Enter>...');
          ReadLn;
        end;
      ExecRestoreScreen :  {restore screen, don't dispose of screen buffer}
        RestoreWindow(1, 1, ScreenWidth, ScreenHeight, False, P);
    end;
  end;
  {$F-}
begin
  UseEmsIfAvailable := False;
  SetSwapMsgOn(False);

  {allocate a screen buffer}
  BufSize := 2*ScreenWidth*ScreenHeight;
  GetMem(P, BufSize);
```

```
    {execute the DIR command}
    Code := ExecDosSwap('DIR', True, @MyExecDosProc, 'JUNK.SWP');
    case Code of
       0 : {ok} ;
      -1 : WriteLn('Insufficient memory to store free list');
      -5 : WriteLn('Unable to allocate EMS or open swap file');
    else
      WriteLn('DOS error ', Code);
    end;

    {free the screen buffer}
    FreeMem(P, BufSize);
  end.
```

Runs the DOS DIR command. This example always swaps to disk and never displays a swapping message.

MyExecDosProc is a user-written action code handler. You could move this procedure, along with the call to ExecDosSwap, to another unit that doesn't use OPEXEC. If you want MyExecDosProc to refer to NoOfBytesSwapped in that case, you'll need to do so through a pointer. Add the following variable declaration to the ExecAccess unit described earlier:

```
var
  NoOfBytesSwappedPtr : ^LongInt;
```

Add the following initialization statement to the main block of the main program:

```
NoOfBytesSwappedPtr := @OpExec.NoOfBytesSwapped;
```

Finally, change the action code handler to

```
WriteLn(NoOfBytesSwappedPtr^, ' bytes are being swapped');
```

See Also

ExecDos (OPDOS) PrepareExecWithSwap

Declaration

```
procedure ExecWithSwap(Path, CmdLine : String);
```

Purpose

DOS shell with swapping to EMS or disk.

Description

This procedure implements one component of the actions taken by ExecDosSwap. You may find it useful if a) you want to allocate the swap file or EMS one time only and leave it allocated throughout your program, or b) you want to gain more control over exactly how much memory is swapped. ExecWithSwap assumes that the swap device has already been validated and opened by a call to PrepareExecWithSwap. You must later call RemoveExecWithSwap to close the swap device.

The parameters to ExecWithSwap are interpreted just like those for the Exec procedure in the DOS unit. Path is the complete pathname of the program to execute. CmdLine is the command line to pass to that program. Note that ExecWithSwap does not find the DOS command processor for you or build an appropriate command line to pass to it (as do ExecDosSwap and ExecDos).

Check the DosError variable for status after ExecWithSwap returns. Error codes are interpreted just like other DOS error codes. See the Turbo Pascal Reference Guide under "Runtime Errors" for more information.

See PrepareExecWithSwap for further information and examples.

See Also

PrepareExecWithSwap RemoveExecWithSwap

PrepareExecWithSwap

Declaration

```
function PrepareExecWithSwap(LastToSave : Pointer;
                             SwapFileName : PathStr) : Boolean;
```

Purpose

Set up for a shell with swapping, returning True if successful.

Description

PrepareExecWithSwap implements one component of the actions taken by ExecWithSwap. You may find it useful if a) you want to allocate the swap file or EMS one time only and leave it allocated throughout the program, or b) you want to gain more control over exactly how much memory is swapped.

LastToSave specifies an address just beyond the last memory location that ExecWithSwap will swap in and out. The first location to save is always the same: midway through the OPEXEC unit. What you choose for LastToSave affects only the size of the swap file or the amount of EMS memory needed, but has no effect on resident overhead during the Exec call.

There are three reasonable values for LastToSave. Specifying the SYSTEM variable HeapOrg tells OPEXEC not to save any part of the heap; this is the best option for programs that make absolutely no use of the heap. Specifying HeapPtr causes OPEXEC to save all allocated portions of the heap. Since only the free list is ignored, this is a good choice for programs that will neither allocate nor deallocate heap space after the ExecWithSwap call returns. Specifying the expression Ptr(Seg(FreePtr^)+$1000, 0) tells OPEXEC to save the entire heap, including the free list. This is the most conservative option, but it may lead to swap files approaching 640K bytes in size.

When ExecDosSwap calls PrepareExecWithSwap, it passes HeapPtr for LastToSave. Before doing so, however, it allocates space on the heap for a copy of the free list. After the child program returns, it restores the free list and deallocates the temporary buffer. In this way ExecDosSwap minimizes the swap file size while still allowing unrestricted use of the heap before and after the exec.

SwapFileName specifies the name and location of the swap file. The requirements for this file match those described in ExecDosSwap.

PrepareExecWithSwap computes the number of bytes to swap and stores the value in NoOfBytesSwapped. If UseEmsIfAvailable is True and sufficient EMS space is available, it allocates the space. Otherwise, it creates the swap file and leaves it open. If neither of these is possible, PrepareExecWithSwap returns False.

Example

```
if not PrepareExecWithSwap(Ptr(Seg(FreePtr^)+$1000, 0), 'JUNK.SWP') then
  WriteLn('Unable to allocate swap space')
else begin
  ExecWithSwap('D:\OPRO\MAKEMENU.EXE', '');
  if DosError <> 0 then
    WriteLn('Exec error ', DosError);
  RemoveExecWithSwap;
end;
```

Shows the proper sequence for calling PrepareExecWithSwap, ExecWithSwap, and RemoveExecWithSwap.

See Also

ExecDosSwap

Declaration

```
procedure RemoveExecWithSwap;
```

Purpose

Deallocate EMS space or erase swap file, whichever was allocated.

Description

RemoveExecWithSwap cleans up after PrepareExecWithSwap. You don't need to call this routine if you're using just ExecDosSwap. OPEXEC also installs an exit handler that automatically calls RemoveExecWithSwap when the program halts.

See PrepareExecWithSwap for further information and examples.

See Also

ExecDosSwap

Declaration

```
procedure SetSwapMsgOn(On : Boolean);
```

Purpose

Turns Swap message on or off.

Description

The swap message is turned on by default. The boolean typed constant NoSwapMsgForEms also controls whether the message appears when swapping to EMS.

OPEXEC's swap message is written to the StdErr device using DOS services. While swapping out occurs during the call to the child process, the message 'Swapping out...' is written at the current cursor location. When the swap is done, OPEXEC writes a carriage return and enough spaces to clear the message. *Important:* In order for this to work correctly, the cursor must have been in column 1 when the message was written.

On return from the child process, OPEXEC writes a <CR><LF> followed by the message 'Swapping in...' and clears it with spaces after the swap is complete.

Example

```
SetSwapMsgOn(False);
```

Turns off the swapping message.

D. OPENHKBD: Enhanced Keyboard Interface

Although enhanced keyboards (the ones with the 12 function keys rather than 10) have quickly become standard equipment on new machines, there are still relatively few programs that fully support them. There are three main reasons for this. First, in order to read all the keys on an enhanced keyboard you have to use special BIOS function calls. Second, before you can use these function calls, you have to be sure that an enhanced keyboard is installed, and there's no standard means of doing so. Third, it is difficult to write programs that work equally well on both "old-style" and enhanced keyboards, because many of the special key combinations recognized by the enhanced keyboard, such as <AltEsc> and <CtrlTab>, are not recognized by the old-style keyboard.

OPENHKBD solves these problems by detecting and activating the enhanced keyboard on machines that have one, and simulating an enhanced keyboard on machines that don't. When OPENHKBD is installed, the enhanced scan codes will be returned by all procedures that read from the keyboard: ReadKey (CRT/OPCRT), ReadKeyWord (OPCRT), and any other routine that uses the standard DOS/BIOS keyboard services. There are no procedures that you need to call here: the unit installs and deinstalls itself automatically, and its operation is completely transparent to any program that uses it.

How OPENHKBD Works

OPENHKBD performs its magic by installing two interrupt handlers. The first takes over interrupt $16, the BIOS keyboard interrupt, and traps all calls to the BIOS requesting keyboard input. If an enhanced keyboard is installed, the interrupt handler will reroute any calls to the standard keyboard functions ($00, $01, and $02) to the corresponding enhanced keyboard functions ($10, $11, and $12).

The second interrupt handler takes over interrupt $9, the hardware interrupt that is generated each time that a key is pressed or released. It checks for certain key combinations that are not normally returned by the BIOS. If one of these combinations is found, it circumvents the BIOS keyboard handler and inserts the appropriate key code (see the table below) into the keyboard buffer.

Key Codes

The following table summarizes the key codes that are returned if a given key or combination of keys is pressed while OPENHKBD is in use. All numbers are in hex and indicate the value that would be returned by a call to the ReadKeyWord routine in OPCRT. The high byte of the word is the scan code, and the low byte is the ASCII character. If you are using the ReadKey function (in CRT or OPCRT) to obtain keyboard input, and the character it returns is #0 (representing the low byte of the key code), a second call to ReadKey would return the scan code (the high byte). Any keys not listed here are returned in the usual manner (see the key code chart in Appendix C of the Turbo Pascal 5.0 Reference Guide, or the quick reference chart in DESKPOP). See the Notes following the table for the key to the special symbols that appear next to some of the key codes.

Key	Plain	Shift	Control	Alt
F11	8500%	8700%	8900%	8B00%
F12	8600%	8800%	8A00%	8C00%
Esc	011B	011B	011B	0100*
Backquote	2960	297E	2960	2900*
Backspace	0E08	0E08	0E7F	0E00*
Tab	0F09	0F00	9400*	A500*

Key	Plain	Shift	Control	Alt	
Left Bracket	1A5B	1A7B	1A1B	1A00*	
Right Bracket	1B5D	1B7D	1B1D	1B00*	
Backslash	2B5C	2B7C	2B1C	2B00*	
Semicolon	273B	273A	----	2700*	
Quote	2827	2822	----	2800*	
Enter	1C0D	1C0D	1C0A	1C00*	
Comma	332C	333C	----	3300*	
Period	342E	343E	----	3400*	
Slash	352F	353F	----	3500*	
Insert	5200	5230	9200*	A200#	
Del	5300	532E	9300*	A300#	
Home	4700	4737	7700	9700#	
End	4F00	4F31	7500	9F00#	
PgUp	4900	4939	8400	9900#	
PgDn	5100	5133	7600	A100#	
Up	4800	4838	8D00*	9800#	
Down	5000	5032	9100*	A000#	
Left	4B00	4B34	7300	9B00#	
Right	4D00	4D36	7400	9D00#	
Pad-Asterisk	372A	!!!!	!!!!	3700#	(numeric keypad keys)
Pad-Minus	4A2D	4A2D	8E00*	4A00#	
Pad-Plus	4E2B	4E2B	9000*	4E00#	
Pad-5	4C00*	4C35	8F00*	9C00#	

Notes:

----	These keystrokes are ignored.
*	These keystrokes are not normally returned by the non-enhanced keyboard.
#	These keystrokes are not normally returned by the enhanced keyboard.
!!!!	These keystrokes control printscreen and print echoing. They cannot simulate the effect of the enhanced keyboard, which has the Asterisk and PrtSc keys separated.
%	F11 and F12 are returned only if they actually exist on the keyboard.

With a few exceptions, the OPENHKBD unit returns results identical to those of the enhanced keyboard BIOS. One difference makes the enhanced keyboard simpler to use in a program: those scan words that normally contain $E0 in the low byte to indicate that the key is specific to the enhanced keyboard (like the dedicated cursor keys) will have the low byte cleared to 0 by OPENHKBD.

Keys marked with # are normally ignored by the enhanced BIOS. By defining these numeric keypad keys when <Alt> is pressed, the ability to enter arbitrary keys on the numeric keypad is disabled. Within applications, the ability to use Alt-shifted arrow keys will generally be more useful. Note, however, that you can still enter arbitrary keys on the numeric keypad by pressing both <Alt> and <LeftShift> simultaneously while entering the number. For example, to enter #196 you would hold down the <Alt> and <LeftShift> keys with your left hand, enter 196 on the numeric keypad with your right hand, then release <Alt> and <LeftShift>.

Note that the quick reference chart in DESKPOP displays all the special key codes returned by OPENHKBD. If you pop up DESKPOP, open the Reference window, and look at the column labeled 'Extended', you'll see that these special keys are displayed in high intensity or white, making it easy to distinguish standard and non-standard key codes.

Miscellaneous Issues

The test used by OPENHKBD to detect the presence of an enhanced keyboard is identical to that used by the EnhancedKbdInstalled function in OPCRT, and it is not foolproof. For a

discussion of the test, see the section on "Enhanced keyboard support" in the introduction to the documentation for OPCRT.

OPENHKBD should generally appear early in a program's USES list, preferably first. This is particularly important for TSR's. If you are using OPENHKBD in a program that also uses OPTSR, OPENHKBD must come before OPTSR in the USES list to insure that interrupt vectors can be restored in the proper order.

Important warning: Because this unit takes over interrupt $9, it can crash the system if SideKick is installed and a TSR that takes over interrupt $9 has previously been loaded after SideKick. (This is why Borland has always recommended that SideKick be loaded last.) You must exercise similar caution in using this unit in programs that must take over interrupt $9 for other reasons. This is not a concern for programs that use OPTSR or OPSWAP, since they automatically disable the code in SideKick that causes problems for interrupt $9 handlers, nor is it usually a concern with SideKickPlus.

It is, however, a concern in programs using OPENHKBD that execute other programs (by calling Exec, ExecDos, ExecDosSwap, etc.). In such a program, you should use the following strategy:

```
{restore keyboard vectors grabbed by OPENHKBD}
RestoreKbdVectors;
{--- execute other program ---}
{reenable OPENHKBD's interrupt handlers}
InitKbdVectors;
```

Following these steps will insure that no problems will occur if SideKick is loaded ahead of your program and the program that you are executing also takes over interrupt $9.

Constants

```
EnableEnhanced : Boolean = True;
```

Setting this variable to False temporarily disables OPENHKBD's interrupt handlers.

Variables

```
HasEnhancedKbd : Boolean;
```

This variable is set to True when an enhanced keyboard has been detected. This variable is interfaced primarily for informational purposes. Altering its value may lead to undesirable results.

Declaration

```
procedure InitKbdVectors;
```

Purpose

Save and setup interrupt vectors taken by OPENHKBD.

Description

This routine is intended to be used only in conjunction with RestoreKbdVectors. It is called automatically by OPENHKBD's initialization code, and it must not be called again unless RestoreKbdVectors has been called first. Calling InitKbdVectors without calling RestoreKbdVectors beforehand will result in a system crash the next time that a key is pressed.

InitKbdVectors is interfaced for one purpose only: to allow you to temporarily restore the interrupt vectors grabbed by OPENHKBD (by calling RestoreKbdVectors) and later reactivate its interrupt handlers. See the section on "Miscellaneous Issues," above, for details.

Declaration

```
procedure RestoreKbdVectors;
```

Purpose

Restores original vectors for INT's $09 and $16.

Description

This routine restores the interrupt vectors that are grabbed by OPENHKBD when a program begins. These vectors are restored automatically when the program ends, but in some cases it may be necessary to restore them manually by calling this procedure. The most common case involves a TSR that has just called DisableTSR to remove itself from memory (see SMACS.PAS for an example), but an equally important case is that of a program that executes other programs (see the section on "Miscellaneous Issues," above).

Once RestoreKbdVectors has been called, you can reactivate the unit's interrupt handlers by calling InitKbdVectors. If you just need to temporarily disable the keyboard remapping functions performed by the interrupt handlers, set the global variable EnableEnhanced to False, then back to True when you want to reenable the interrupt handlers.

Example

```
uses OpEnhKbd, OpTsr;
...
if DisableTSR then
  RestoreKbdVectors
else
  {display error message--not safe to unload the TSR} ;
```

Here RestoreKbdVectors is called as soon as it is certain that the TSR will be removed from memory. Note that OPENHKBD must appear *before* OPTSR in the USES statement to insure that the interrupt vectors grabbed by both units are restored in the proper order.

See Also

DisableTSR (OPTSR)

11. Interrupts and TSRs

A. Overview

Surely two of the most daunting tasks faced by PC programmers are the writing of safe Interrupt Service Routines (ISRs for short) and--perhaps the ultimate challenge--rock solid programs that Terminate and Stay Resident (hence the term TSR) without causing trouble. Although **Object Professional** can't guarantee your resident programs to be bug-free or insure that your ISRs don't use too much stack space, it does provide a host of routines to help get you where you need to go, fast.

The primary purpose of OPINT is to manage your ISRs for you. When you call DOS (or the SetIntVec routine in the DOS unit) to take over an interrupt vector, it asks you for the number of the interrupt and the address of your ISR. The InitVector function in OPINT does too, but it also asks you to specify a handle (much like a DOS file handle). This handle is what allows it to keep track of things for you, and once you've called InitVector you can usually forget about your ISR. If you do need to restore an interrupt vector yourself, you can just call RestoreVector with the handle, and OPINT will supply DOS with the correct interrupt number and the address of the old service routine. But if your program ends normally, or even if it aborts with a runtime error, you don't need to worry about the interrupt vectors you've captured, because OPINT will automatically restore all of them for you.

Turbo Pascal provides a great convenience in the form of the "interrupt procedure," which saves and restores all the CPU registers for you, initializes the DS (Data Segment) register, and sets up a stack frame so that you can treat these registers as parameters. This in itself is sufficient to allow you to write simple ISRs without resorting to assembly language, but in many cases it isn't nearly enough. So the secondary purpose of the OPINT unit is to provide assistance in the actual writing of your ISRs, especially large ones that need lots of stack space. It contains routines to allocate and deallocate alternate stacks, to switch stacks from inside your ISR, to quickly return from the interrupt no matter how many procedures you've called, to "chain" to the previous interrupt handler, and more.

If writing and debugging ISRs is like climbing jagged rocks, developing a professional-quality TSR can be likened to scaling a glacier. The light is blinding, the surface slippery. But as the wealth of memory resident demonstration programs included in the package demonstrates, **Object Professional** can help you make quick work of this mountainous task.

You've probably never felt the need to write a pop-up "Hello World!" program, but if you did, how would you do it? How many lines of source code would it take? This is how you could do it using **Object Professional**:

```
{$S-,I-,R-}
{$M 2048,0,655360}
program HelloWorld;
uses Dos, OpInt, OpTsr;
const
  HotKey = $080F;              {code for Alt-Tab}
  WaitForDos = True;           {don't pop up in the middle of DOS}

  {$F+}
  procedure PopupEntryPoint(var Regs : Registers);
  begin
    WriteLn('Hello world!');
  end; {$F-}
```

```
begin
  {define the popup}
  if DefinePop(HotKey, PopupEntryPoint, Ptr(SSeg,SPtr), WaitForDos) then begin
    WriteLn('Pop up loaded, press <Alt><Tab> to activate.');
    {Make popup routines active.}
    PopupsOn;
    {try to go resident}
    StayRes(ParagraphsToKeep, 0);
  end;

  {if we get here we failed}
  WriteLn('Unable to go resident');
end.
```

You won't want to pass this program around to your friends, of course, but what can you expect for a mere 30 lines of source code?

As you can see, writing TSRs with **Object Professional** is remarkably easy, and what you get for your efforts are solid programs that won't pop up in the middle of DOS if you don't want them to; that won't destroy the interrupted program's stack; that can pop up on virtually any combination of keystrokes you might want; that won't load themselves twice (if you're willing to make a couple of simple procedure calls before going resident); that won't abort due to DOS critical errors; and that can easily be unloaded from memory if it's safe to do so. (All of the **Object Professional** demonstration programs that terminate and stay resident offer this option, by the way.) There's even a facility that allows your TSRs to communicate with each other and an error handler that tries to prevent your most innocent runtime errors from violently crashing the system.

A couple of years ago, when we introduced Turbo Professional 4.0, the OPTSR unit (then called TPTSR) represented state-of-the-art technology. The state of the art changes rapidly in this business, of course, and today even a measly 30K can be considered too much memory to devote to a single TSR. Enter the "swappable TSR." Unlike traditional TSRs, a swappable TSR takes up only a few kilobytes of memory when it is not in use, no matter how big it is. Briefly described, the trick is this. Before going resident, a swappable TSR writes most of its code and data out to disk (or EMS), then gives as much memory as it can back to DOS. We'll refer to the region of memory written to disk/EMS as "the swap region." Once the TSR has gone resident, all that is left in memory is a tiny "kernel," consisting mainly of interrupt handlers which monitor the state of the system and watch for requests to activate the TSR. When a request comes in, the kernel writes the current contents of the swap region out to disk/EMS, reloads the previously swapped code and data into its place, and then calls the appropriate routine in the now reconstructed program.

It's a fairly simple concept in theory, but one that's extremely difficult to implement. Fortunately, we've done all the dirty work for you, by marrying the TSR technology in OPTSR to the swapping technology in OPEXEC. As a result, writing a swappable TSR using our OPSWAP and OPSWAP1 units is only slightly more difficult than writing a regular TSR using OPTSR. Here, for example, is a swappable version of our 'Hello World!' TSR, which is now divided into two parts, a tiny main program:

```
{$S-,I-,R-}
{$M 2048,0,655360}
program HelloWorld;
uses HWmain, OpSwap;
begin
  HelloWorldMain;
end.
```

and a unit that contains the core of the application:

```
{$S-,I-,R-}
unit HWmain;
interface

uses
  OpInline, OpSwap1;
  procedure HelloWorldMain;

implementation

const
  HotKey = $080F;             {code for Alt-Tab}
  Swap1 = 'HELLO1.SWP';
  Swap2 = 'HELLO2.SWP';

  {$F+}
  procedure PopupEntryPoint;
  begin
    WriteLn('Hello world!');
  end;
  {$F-}

  procedure HelloWorldMain;
  begin
    {disable swapping message if swapping to EMS}
    SetSwapMsgOn( not WillSwapUseEms(ParagraphsToKeep) );
    {define the popup}
    if DefinePop(HotKey, PopupEntryPoint, Ptr(SSeg,SPtr)) then begin
      WriteLn('Pop up loaded, press <Alt><Tab> to activate.');
      {Make popup routines active.}
      PopupsOn;
      {try to go resident}
      StayResSwap(ParagraphsToKeep, 0, Swap1, Swap2, True);
    end;

    {if we get here we failed}
    WriteLn('Unable to go resident');
  end;

end.
```

If you look closely, you'll see that most of the differences between the two programs are minor and cosmetic. The big difference is one that you can't see by looking at the source code: when resident, the original HELLO program uses about 12K, while the swappable version uses about 6K. That sounds like a small difference, and it is, but only because our test case is small. To get a better idea of how powerful OPSWAP and OPSWAP1 are, consider that our demonstration program DESKPOP, which also uses just 6K of memory, would use about 400K if it were written with OPTSR!

The OP8087 unit is intended primarily to help you write resident programs that use the numeric coprocessor (an 80x87 chip) for real number calculations. The Save8087 procedure allows you to save the coprocessor's registers when your TSR pops up, and Restore8087 to restore them on exit. OP8087 also offers two routines to control how the 80x87 handles errors. The Exceptions8087 procedure lets you disable exception interrupts or reenable them. The Error8087 function returns the current coprocessor status, allowing you to determine whether a calculation error occurred in the preceding sequence of instructions. OP8087 also supports Turbo Pascal's floating point emulator.

B. OPINT: ISR Management

As indicated at the start of this chapter, the OPINT unit serves three important functions, the most important of which is ISR (Interrupt Service Routine) management.

By that we mean simply that OPINT keeps track of your interrupt handlers for you. The normal procedure for capturing an interrupt vector in Turbo Pascal is to 1) call the GetIntVec routine in the DOS unit, 2) save the current vector, 3) call SetIntVec to point the vector to your ISR, and 4) restore the old vector when your program ends. Step 4 is the hard one, because a program can end for any number of reasons and at any number of points. In many cases, OPINT lets you reduce these four steps to one:

```
const MyIsrHandle = 16;
...
if InitVector(IntNumber, MyIsrHandle, @MyIsr) then {};
```

OPINT will save the current vector for you, change it to point to MyIsr, and restore the old vector when your program ends.

OPINT can keep track of up to 20 (MaxISRs) ISRs, far more than any program is likely to have, and each one must be assigned a unique handle. A handle is simply a number used to identify a particular ISR, and in most programs you can use any number between 1 and 20. Be advised, however, that two other units in **Object Professional** that use OPINT reserve particular handles for their own use. OPTSR greedily demands numbers 1-11, and OPMACRO uses #12. Since the number of ISR handles reserved for other units may grow in the future, we recommend using only handle numbers 16 and up, especially in your TSRs.

The other main function of OPINT is to provide routines that simplify the process of writing interrupt service routines in Pascal. All of these routines depend on two things: 1) Turbo Pascal's "interrupt procedures " and 2) variables of a data type defined in OPINT, IntRegisters. If you've studied the Turbo Pascal manual's discussion of interrupt procedures, you know that it allows you to treat the CPU registers as parameters by declaring the interrupt procedure thus:

```
procedure MyIsr(Flags, CS, IP, AX, BX, CX, DX, SI, DI, DS, ES, BP : Word);
  interrupt;
begin
end;
```

To use the procedures we're talking about, you would simply change this declaration to:

```
procedure MyIsr(BP : Word); interrupt;
var
  Regs : IntRegisters absolute BP;
begin
end;
```

Doing so buys you two things. First, you can now refer to the "half registers" (AH, AL, BH, BL, etc.) using "Regs.AH", "Regs.AL", and so on. Second, you can now treat the various registers as a group, and pass all of them together as a single parameter to a procedure.

Now, assuming that you've set up your ISR as we've described, you can use the five routines that we've been preparing to talk about. IntReturn lets you return from an interrupt from a procedure other than your ISR. For example, you could call a routine like this one:

```
procedure MyIsrHelper(var Regs : IntRegisters);
begin
  {--- do something ---}
  IntReturn(Regs);
end;
```

ChainInt lets you end your interrupt and pass control to the old interrupt vector like so:

```
ChainInt(Regs, IsrArray[MyIsrHandle].OrigAddr);
```

SwapStackAndCallNear and SwapStackAndCall let you switch to an alternate stack and call another procedure from inside your ISR. When the procedure that you have called returns control, the old stack is restored. And EmulateInt lets you call the old interrupt vector from inside your ISR without exiting. For example:

```
EmulateInt(Regs, IsrArray[MyIsrHandle].OrigAddr);
```

Finally, OPINT provides two functions that make it a bit easier to allocate and deallocate alternate stacks. AllocateStack allocates a stack (just an array, really) of a specified size on the heap, and returns a pointer to the end of the array, since the machine stack grows downward. Its companion procedure, DeallocateStack, disposes of a stack allocated in this fashion.

Reminder: As indicated in the Turbo Pascal manual (Inside Turbo Pascal section, under Writing Interrupt Procedures), Turbo Pascal's I/O routines and memory management routines are not reentrant, and cannot be used in an interrupt service routine. DOS isn't reentrant, so DOS services can't be used. And, finally, many **Object Professional** routines are not reentrant. If you need to be certain, check the source code for the routine: any routine that modifies a global variable declared in an **Object Professional** unit is probably not reentrant.

Constants

```
MaxISRs = 20;
```

The maximum number of interrupt service routines that can be managed by OPINT.

Types

```
Dummy5 = array[1..5] of Word;
IntRegisters =
  record
    case Byte of
      1 : (BP, ES, DS, DI, SI, DX, CX, BX, AX, IP, CS, Flags : Word);
      2 : (Dummy : Dummy5; DL, DH, CL, CH, BL, BH, AL, AH : Byte);
  end;
```

A record whose structure mirrors the contents of the stack frame in an interrupt procedure.

```
IsrRecord =
  record
    IntNum : Byte;
    OrigAddr, NewAddr : Pointer;
    Captured : Boolean;
  end;
```

A record containing information about a particular interrupt service routine. IntNum is the vector that was taken over. OrigAddr is the address of the previous ISR. NewAddr is the address of the current ISR. And Captured is used internally to determine whether or not the vector has already been restored.

Variables

```
IsrArray : array[1..MaxISRs] of IsrRecord;
```

The global array of records containing data about the ISRs being managed by OPINT.

AllocateStack / ChainInt

Declaration

```
function AllocateStack(var P : Pointer; SizeInBytes : Word) : Boolean;
```

Purpose

Allocates a stack of size SizeInBytes.

Description

This function provides a convenient means of allocating alternate stacks for interrupt service routines and popups. SizeInBytes is the size of the stack to be allocated, and cannot be greater than 65,515. If a stack of the desired size can be allocated, AllocateStack returns a pointer to the end of it in P. If not, the function returns False.

Important: To deallocate a stack allocated by AllocateStack, you *must* use DeallocateStack.

Example

```
if not AllocateStack(MyStack, MyStackSize) then
  Halt(1);
```

Here AllocateStack is called to allocate a stack of MyStackSize bytes. If there is insufficient memory available on the heap, the program aborts.

See Also

DeallocateStack

Declaration

```
procedure ChainInt(var Regs : IntRegisters; JumpAddr : Pointer);
```

Purpose

Restores stack, registers from Regs and 'jumps' to JumpAddr.

Description

This inline macro allows you to restore the CPU registers pushed onto the stack at the beginning of an interrupt procedure, then pass on the interrupt to the ISR that serviced the vector before you grabbed it. Unlike the JumpToOldIsr macro in OPINLINE, ChainInt is intended primarily for ISRs written in Pascal, rather than inline code.

Another difference between ChainInt and JumpToOldIsr is that ChainInt may be called from a procedure other than the actual interrupt procedure. In order to do so, however, the variable passed as the Regs parameter *must* have been passed to the second procedure as a VAR parameter. In addition to reloading the CPU registers based on the contents of the Regs parameter, ChainInt also has to reset the stack, so it cannot work with a mere copy of the original IntRegisters variable.

Examples

```
procedure MyIsr(BP : Word); interrupt;
var Regs : IntRegisters absolute BP;
begin
  {-- something --}
  ChainInt(Regs, IsrArray[MyIsrHandle].OrigAddr);
end;
...
if not InitVector($09, MyIsrHandle, @MyIsr) then {error};
```

Correct. ChainInt will restore the registers and "jump" to the previous ISR.

```
ChainInt(Regs, IsrArray[MyIsrHandle].OrigAddr^);
```

Incorrect! The '^' following OrigAddr is not needed.

See Also

EmulateInt JumpToOldIsr (OPINLINE)

Declaration

```
procedure DeallocateStack(P : Pointer);
```

Purpose

Frees an allocated stack.

Description

This routine allows you to easily dispose of a stack allocated with AllocateStack. Note that the size of the stack is not required as a parameter, since AllocateStack will have already stored that information (just above the top of the stack) for DeallocateStack's use.

Important: Don't use DeallocateStack to dispose of a stack not allocated by AllocateStack.

Example

```
if not AllocateStack(MyStackPtr, MyStackSize) then {error};
...
DeallocateStack(MyStackPtr);
```

See Also

AllocateStack

Declaration

```
procedure EmulateInt(var Regs : IntRegisters; IntAddr : Pointer);
```

Purpose

Emulates an interrupt.

Description

This routine simulates an actual interrupt by filling the CPU registers with the values in Regs, clearing interrupts, pushing the flags, and calling FAR to IntAddr. Like the CallOldIsr macro in OPINLINE, it thus allows you to call the ISR that serviced a particular interrupt vector before you grabbed it. Unlike CallOldIsr, however, EmulateInt is intended primarily for ISRs written in Pascal, rather than inline code.

Another difference between EmulateInt and CallOldIsr is that EmulateInt may be called from a procedure other than the actual interrupt procedure. In order to do so, however, the variable passed as the Regs parameter *must* have been passed to the second procedure as a var parameter. Although EmulateInt does not reset the stack, as ChainInt does, it does modify the contents of the Regs parameter.

A second common use of EmulateInt is to call a routine via a command entry pointer. See the documentation for the OPTSR unit for details.

Examples

```
if not InitVector($09, MyIsrHandle, @MyIsr) then {error};
...
procedure MyIsr(BP : Word); interrupt;
var Regs : IntRegisters absolute BP;
begin
  {-- something --}
  EmulateInt(Regs, IsrArray[MyIsrHandle].OrigAddr);
end;
```

Correct.

```
EmulateInt(Regs, IsrArray[MyIsrHandle].OrigAddr^);
```

Incorrect! The '^' after OrigAddr is not needed.

See Also

CallOldIsr (OPINLINE) ChainInt

InitVector / InterruptsOff / InterruptsOn

Declaration

```
function InitVector(IntNumber, Handle : Byte; UserRoutine : Pointer) : Boolean;
```

Purpose

Sets up an interrupt service routine.

Description

The central routine in the OPINT unit, InitVector allows you to change an interrupt vector (IntNumber) to point to an interrupt service routine that you have written (pointed to by UserRoutine). The Handle is the number that you want to use to identify the ISR in question. (See the overview at the start of this section for a discussion of handles.) Once you've called InitVector, all you need to worry about is servicing the interrupt. When your program ends, the old vector will be restored for you automatically, even if a runtime error occurred. If you need to restore the vector before the program ends, you can do so with RestoreVector or RestoreAllVectors. InitVector returns False only if the Handle specified is greater than MaxISRs or it is already in use.

Examples

```
if not InitVector($09, MyIsrHandle, @MyIsr) then {handle already in use} ;
```

Here InitVector is used to change interrupt vector $09 to point to MyIsr.

```
RestoreVector(MyIsrHandle);
Code := ExecDos('', True, NoExecDosProc);
if not InitVector($09, MyIsrHandle, @MyIsr) then {shouldn't occur};
```

In this example, RestoreVector is called to restore the old interrupt $09 handler before dropping into a DOS shell, and InitVector is called afterward to grab back the vector.

See Also

RestoreVector RestoreAllVectors

Declaration

```
procedure InterruptsOff;
```

Purpose

Turn interrupts off.

Description

This tiny inline macro turns interrupts off.

Declaration

```
procedure InterruptsOn;
```

Purpose

Turn interrupts on.

Description

This tiny inline macro turns interrupts on.

Example

```
procedure MyIsr(BP : Word); interrupt;
var Regs : IntRegisters absolute BP;
begin
  InterruptsOff;
  ...
  InterruptsOn;
end;
```

Here interrupts are turned off at the beginning of the ISR, and turned on again at the end.

Declaration

```
procedure IntReturn(var IntRegs : IntRegisters);
```

Purpose

Return from interrupt.

Description

This routine lets you return from an interrupt from almost anywhere in your program, provided that the current procedure has been passed a var parameter of type IntRegisters by the actual ISR. Like ChainInt, IntReturn *cannot* work with a mere copy of the original IntRegisters variable.

IntReturn is needed only when the stack has been changed, presumably using SwapStackAndCall or SwapStackAndCallNear, or when you need to return from an interrupt while within a procedure called by the actual ISR.

Example

```
procedure NonIsr(var Regs : IntRegisters);
begin
  {-- something --}
  if NeedToReturnFast then
    IntReturn(Regs);
  {-- something --}
end;

procedure MyIsr(BP : Word); interrupt;
var Regs : IntRegisters absolute BP;
begin
  if Regs.AX = MagicValue then
    NonIsr(Regs)
  else begin
    ...
  end;
end;
```

MyIsr calls NonIsr, passing Regs as a parameter, and NonIsr uses IntReturn to return from the interrupt. Note that the Regs parameter to NonIsr *must* be a var parameter.

See Also

ChainInt
SwapStackAndCall
SwapStackAndCallNear

Declaration

```
procedure RestoreAllVectors;
```

Purpose

Restores all captured interrupt vectors.

Description

This routine allows you to restore all interrupt vectors you have captured. RestoreAllVectors is called for you automatically when your program ends.

Example

```
if not InitVector($09, MyIsrHandle, @MyIsr) then {error}
if not InitVector($16, MyOtherIsrHandle, @MyOtherIsr) then {error}
...
RestoreAllVectors;
```

See Also

RestoreVector

RestoreVector / SendEOI

Declaration

```
procedure RestoreVector(Handle : Byte);
```

Purpose

Restores an interrupt vector to its original value.

Description

This routine allows you to restore a vector previously grabbed using a call to InitVector. Since any interrupt vectors that you have grabbed are restored for you automatically when your program ends, the primary use for RestoreVector is to restore a vector temporarily. See the example shown below.

Example

```
if not InitVector($09, MyIsrHandle, @MyIsr) then {handle already in use};
...
RestoreVector(MyIsrHandle);
Code := ExecDos('', True, NoExecDosProc);
if not InitVector($09, MyIsrHandle, @MyIsr) then {shouldn't happen};
```

Restore the old interrupt $09 handler before dropping into a DOS shell.

See Also

InitVector RestoreAllVectors

Declaration

```
procedure SendEOI;
```

Purpose

Send an End Of Interrupt signal to the Programmable Interrupt Controller.

Description

This inline macro sends an EOI (End Of Interrupt) signal to the Programmable Interrupt Controller (PIC for short). It should be used only in interrupt handlers that are servicing hardware interrupts such as $8 (clock tick) or $9 (keyboard).

Examples

```
procedure Int9(BP : Word); interrupt;
var
  Regs : IntRegisters absolute BP;
  B : Byte;
begin
  {get the scan code for the key press/release}
  B := Port[$60]; {note that high bit will be set if key was released}
  ...
  if not NeedToEatKeyStroke then
    ChainInt(Regs, OldInt9)
  else begin
    {reset the keyboard}
    B := Port[$61];
    Port[$61] := B or $80;
    Port[$61] := B;
    {signal end of interrupt}
    SendEOI;
  end;
end;
```

A fairly realistic application of SendEOI--an interrupt $9 handler that needs to "eat" keystrokes under certain conditions.

Declaration

```
procedure SwapStackAndCall(Routine, SP : Pointer; var Regs : IntRegisters);
```

Purpose

Switches to stack designated by SP and calls Routine with Regs as a parameter.

Description

This routine allows you to switch to an alternate stack (pointed to by SP) and call a procedure, passing Regs as a parameter. The Routine to be called should be of the form

```
{$F+}
procedure MyProc(var Regs : IntRegisters);
begin
  ...
end;
```

Very important: MyProc must be a FAR call from the current ISR. If it is not, you should use SwapStackAndCallNear instead.

Examples

```
{$F+}
procedure NonIsr(var Regs : IntRegisters);
begin
  ...
  if NeedToReturnFast then
    IntReturn(Regs);
  ...
end;
{$F-}

procedure MyIsr(BP : Word); interrupt;
var Regs : IntRegisters absolute BP;
begin
  if Regs.AX = MagicValue then
    SwapStackAndCall(@NonIsr, MyStackPtr, Regs)
  else begin
    ...
  end;
end;
```

Correct. SwapStackAndCall will switch to the stack pointed to by MyStackPtr and call NonIsr, passing Regs. Notice that NonIsr has been compiled under the FAR model.

```
SwapStackAndCall(@NonIsr, MyStackPtr^, Regs)
```

Incorrect! The '^' after MyStackPtr is not needed.

See Also

SwapStackAndCallNear

SwapStackAndCallNear

Declaration

```
procedure SwapStackAndCallNear(Routine : Word; SP : Pointer;
                               var Regs : IntRegisters);
```

Purpose

Switches to stack designated by SP and calls Routine with Regs as a parameter.

Description

This routine allows you to switch to an alternate stack (pointed to by SP) and call a procedure, passing Regs as a parameter. The Routine to be called should be of the form

```
{$F-}
procedure MyProc(var Regs : IntRegisters);
begin
  {-- something --}
end;
```

Important: MyProc *must* be a NEAR call from the current ISR. If it isn't, use SwapStackAndCall instead.

Example

```
{$F-}
procedure NonIsr(var Regs : IntRegisters);
begin
  ...
  if NeedToReturnFast then
    IntReturn(Regs);
  ...
end;

procedure MyIsr(BP : Word); interrupt;
var
  Regs : IntRegisters absolute BP;
begin
  if Regs.AX = MagicValue then
    SwapStackAndCallNear(Ofs(NonIsr), MyStackPtr, Regs)
  else begin
    ...
  end;
end;
```

Correct. SwapStackAndCall will switch to the stack pointed to by MyStackPtr and call NonIsr, passing Regs. Notice that NonIsr has been compiled under the NEAR model.

```
SwapStackAndCallNear(Ofs(NonIsr), MyStackPtr^, Regs)
```

Incorrect! The '^' after MyStackPtr is not needed.

See Also

SwapStackAndCall

C. OPTSR: TSR Management

The main purpose of the OPTSR unit is to provide a TSR management system that is easy to use and solid as a rock. By following a few simple guidelines, you should be able to write reliable memory resident programs just as easily as non-resident ones. But before we go into the details of how to do it, let's define a few terms.

A TSR (Terminate and Stay Resident) is a program that, shortly after being loaded, returns control to the operating system (DOS) while remaining in memory. Every TSR contains one or more *popup* routines, which are to be called by the *popup dispatcher* when a particular event occurs. The job of the popup dispatcher may be divided into roughly six parts: 1) monitor the system to watch for events of interest; 2) when such an event occurs, determine which popup needs to be called; 3) based on what is known about the popup, determine whether or not it can be invoked safely; 4) before calling the popup, save the state of the system and take all possible precautions to insure the safety of the popup; 5) call the popup; 6) restore the state of the system. (Note: The order of events described here and in the following paragraph has been intentionally distorted in order to make this complex process easier to understand.)

Now let's look at a simple example, the "Hello World!" program shown at the start of the chapter, so we can see these principles in action. The popup dispatcher monitors the keyboard, waiting for the user to press the <Tab> key at a time when the <Alt> key has already been pressed. When this event occurs, the dispatcher checks its tables and sees that the PopupEntryPoint procedure needs to be called, and that it needs access to DOS services (WaitForDos is True). The dispatcher then checks to be sure that these services can be accessed safely and, assuming that they can, proceeds to create a stable environment for the popup to run in. Finally, PopupEntryPoint is called. Once it has finished, it returns control to the dispatcher, which then restores the state of the system, returns the CPU to the task it was working on when <Alt><Tab> was pressed, and goes back to monitoring the system.

On to the details, which we'll divide into topics.

Number and Kinds of Popups

By default, OPTSR allows you to define as many as 8 (MaxPopups) popup routines in any one program, each of which falls into one of two categories. The kind you will probably use most frequently is a popup associated with a *hotkey*--more on it in a minute. The other kind of popup is not invoked as a result of any particular event. Instead, you can invoke it whenever you want by calling the SetPopTicker procedure with the handle assigned to the popup (see below) and a clock tick count (TimeOut). This tells the popup dispatcher to start watching for a good opportunity to call the popup. If TimeOut clock ticks go by without such an opportunity, the popup will not be called. For a good example of such a popup, see PTIME.PAS, which uses one to display an on-screen clock, to sound alarms, and to play back SMACS macros at designated times.

Hotkeys

The more conventional popup is one that is invoked when a specified combination of keys is pressed simultaneously, a combination referred to collectively as a hotkey.

In **Object Professional**'s terms, a hotkey is actually a single key that is pressed while a particular combination of shift keys is already being pressed. Besides the regular shift keys, <LeftShift> and <RightShift>, the <Alt> and <Ctrl> keys are also treated as shift keys for this purpose. To specify the hotkey for your popup, you need to construct a word value that is bit-mapped as follows:

Bit #	Value	Meaning
15-12	$0000	unused -- must be zero
11	$0800	Alt
10	$0400	Ctrl
9	$0200	Left shift
8	$0100	Right shift
7-0	$00xx	Scan code for main key

For example, let's say that you want your popup to activate when <Alt>, <LeftShift>, and <Z> are pressed. The scan code for <Z> is $2C (see the scan code chart in Appendix C of the Turbo Pascal Reference Guide, or look at the quick reference chart in DESKPOP)--that gives us the value for the low byte, bits 0-7. If we add to that the values for <Alt> and <LeftShift> we get $0800 + $0200 + $002C = $0A2C. (The calculator in DESKPOP is quite handy for doing hexadecimal arithmetic like this, by the way.)

More on Hotkeys

Two other points to keep in mind concerning hotkeys. First, if it wasn't obvious from the previous paragraph, it is not possible to define a popup that is activated by a single keystroke. One of the four shift keys must be specified--if the high byte of the hotkey code is 0, the popup will never be called. It is possible, however, to treat one of the shift keys as the main key. For example, to pop up on <Ctrl> <LeftShift> you could specify either $042A or $021D, depending on which of the two keys was to be pressed first. Second, you may have more than one hotkey per popup. See the entry for the AddHotKey function later in the chapter.

Popup Handles

When you call DefinePopProc to define a popup that is invoked as a result of a call to SetPopTicker, it appears to ask for a "Handle". Unlike the OPINT unit, which allows you to assign your own handles, OPTSR assigns them for you. So all you need to do is declare a global byte variable--say, "MyPopupHandle"--and pass it as the parameter. If any handles are still available, DefinePopProc will assign MyPopupHandle a value and return True. When you are ready to invoke your popup, you would then say something like:

```
SetPopTicker(MyPopupHandle, 18);
```

which means "during the next 18 clock ticks, look for a good opportunity to call the popup associated with MyPopupHandle." Popups activated by hotkeys are also assigned handles, but you don't need to know what they are since they are for housekeeping purposes only.

Popup Routines

All popups written with OPTSR, whether they are activated by the press of a hotkey or by a call to SetPopTicker, should be of the form

```
{$F+}
procedure PopupShell(var Regs : Registers);
begin
  {-- body of the popup --}
end;
```

Both the {$F+} directive and the declaration of the Registers parameter are necessary. If either is missing, you will get a compiler error on your call to DefinePop/DefinePopProc.

Please note that, although Regs is a VAR parameter (to conserve stack space), you should *never* alter it. Regs contains the values of the CPU registers just before your popup was activated, and these values will be put back into the registers when your popup exits. Although

it is unlikely that you will ever need the information contained in the Regs parameter, it is always provided in case you do.

Stacks

Each popup must also have its own stack to work with. This means that the StackPtr parameter passed to DefinePop/DefinePopProc must be different for each popup in the program. Most TSRs consist of a single popup, and in these cases you can almost always use "Ptr(SSeg, SPtr)" as the parameter. But programs that contain multiple popups (such as SMACS.PAS) must allocate space for alternate stacks, either by declaring an array as a global variable or by calling the AllocateStack routine in OPINT, which allocates an array on the heap. In either case, the StackPtr parameter must point to the last element of the array rather than the first. (Note that AllocateStack always returns a pointer to the end of the array.)

Waiting for DOS

Most popups need access to DOS services , but some do not. If you are uncertain, you should play it safe and assume that your popup does. But if you know for a fact that it doesn't (e.g., if it does something simple like draw text on the screen using the FastWrite routine in OPCRT), you can pass "False" to DefinePop as the DosWait parameter. This will allow you to pop up in many situations where you otherwise could not.

"DOS services," by the way, covers a broad range of territory. All disk I/O, most notably, requires DOS's assistance, but so do lots of other things, some of which might surprise you. The GetDate and GetTime routines in the DOS unit, for example, use DOS calls. Again, when in doubt, assume that DOS is needed.

For further details, see the section on "Playing It Safe," below.

Before Going Resident

If the non-resident portion of your program captures any interrupt vectors not needed by the resident portion, they must be restored before going resident. There are a number of vectors captured by the SYSTEM unit that fall into this category, notably the ones used by the overlay manager and the floating point emulator, but OPTSR restores these for you automatically.

Going Resident

OK, you've defined your popups, now what do you do? In many cases, you would simply call the StayRes procedure like so:

```
StayRes(ParagraphsToKeep, 0);
```

ParagraphsToKeep determines how much memory your program is currently using, so the above statement would instruct StayRes to release everything you aren't already using to DOS. Often this is the right thing to do, but if your popups need to allocate space on the heap later on, there won't be any memory left to allocate. So, assuming that you wanted to set aside 4000 bytes (250 paragraphs) of heap space for your popups, you'd do this instead:

```
StayRes(ParagraphsToKeep+250, 0);
```

For an example of a TSR that allocates heap space after going resident, see SMACS.PAS.

Important: It is not necessary to set an especially low value for a resident program's maximum heap size (using the {$M} directive). On the contrary, if you need to allocate things on the heap after going resident, you should use a very high number, and let StayRes move the heap downward for you. The ParasToKeep parameter to StayRes must not request more heap space than is available (i.e., ParasToKeep may not exceed MaxParagraphsToKeep).

Running Your TSR

In case you didn't know it already, TSRs must be loaded from the DOS command line. So, if you are accustomed to compiling your programs from inside Turbo Pascal's IDE (Integrated Development Environment) and then immediately running them, you'll have to make some changes in your programming habits. Be aware, too, that most debuggers cannot be used to debug TSRs, including Borland's Turbo Debugger (versions 1.0 and 1.5) and the debugger built into Turbo Pascal's IDE. The debugger that we use for TSRs is Periscope, and we highly recommend it.

I/O Errors and DOS Critical Errors

One of the things that the dispatcher does before it calls a popup that needs access to DOS services is to install a *DOS critical error handler* to trap the kind of errors that cause DOS to display the familiar "Abort, Retry, Ignore" message. These are bad news in any program, but they can mean disaster for a TSR. The error handler installed by the dispatcher simply tells DOS to ignore the error (under DOS 2.x) or to fail the call (under DOS 3.0 or higher). This will get you back into your program, but under DOS 2.x the error condition will still exist, so you have to be prepared to deal with it. How? At the top of your program, add the {$I-} directive, so the runtime library won't try to abort when an I/O error occurs. After each I/O operation that could conceivably cause an I/O error, add a call to IoResult. If it returns anything other than 0, either an I/O error or a DOS critical error occurred.

Runtime Errors

Runtime errors can be just as deadly, of course. OPTSR contains an error handler that attempts to salvage the system when runtime errors occur, but this isn't always possible. The good news is that it is *usually* possible if your popup uses an alternate stack (in the data segment or on the heap), so you would be well advised to use one while developing your program. (A finished TSR should not generate runtime errors. If it does, it isn't finished.) Note that this error handler does not display any error messages, and if it succeeds in its salvage operation, Turbo's error handler, which does display error messages, will never be reached. So if you need to see what error occurred you'll need to write your own error handler by following the steps outlined in the Turbo Pascal manual.

Using OPTSR with OPWINDOW and Related Units

When writing a TSR that uses OPWINDOW or any of the many units that depend on it, keep in mind that all of the window-based objects make liberal use of the heap, especially if they can be moved or resized. These objects do most of their memory allocation when they are initialized, but in some cases the window stack manager must allocate memory later in order to perform a particular function (zooming or resizing the window, for example).

In general, this means you should 1) initialize all window-based objects before going resident, if possible, 2) check scrupulously for out-of-memory errors before and after going resident, and 3) set aside at least a few hundred extra bytes of heap space when you go resident. For example:

```
StayRes(ParagraphsToKeep+25, 0);
```

This would ensure that at least 400 (25*16) bytes will be available on the heap after going resident. See the earlier section on "Going Resident" for a discussion of issues related to the allocation of heap space while resident.

Of course, the same basic guidelines described here apply to the use of any unit, object, or routine that needs to allocate heap space--OPSORT, for example.

Stack and Heap Usage

One difficult task that OPTSR can't do for you is determine how much stack or heap space a particular popup requires. Our best advice is this. During the development stages, allocate more stack and heap space than you think you could possibly need, and don't worry about memory usage. When your program is mostly finished, use the TPSTACK unit found in the BONUS directory to start getting accurate measurements of heap and stack usage. Although TPSTACK is designed primarily for use in non-resident programs, it can be adapted for use in TSRs if necessary. See the documentation for TPSTACK for instructions.

TSRs and 8087 Reals

If you are writing a TSR that uses 8087 reals, you must be sure to save and restore the state of the math coprocessor within your popup. The TSR manager leaves this job to you because it doesn't know if it is necessary. It does, however, ensure that the interrupt vectors used by the floating point emulator are set properly on entry to your popup. The OP8087 unit provides all the routines you will need for this task. For details, see the documentation for OP8087.

TSRs and EMS

A similar need arises in TSRs that need to use EMS memory while popped up. The OPEMS unit provides the routines that you need to save and restore the state of the Expanded Memory Manager so that you can access EMS memory safely. For details, see the entries for the SaveEmsContext and RestoreEmsContext routines in the documentation for OPEMS.

TSRs and Overlays

Writing a TSR that uses Borland's overlay manager also requires extra work on your part. The following program (based on Borland's OVRDEMO.PAS) illustrates the basic steps involved:

```
{$V-,S-,I-,R-,F+,O+}
{$M 4096,0,650000}
program OvrDemo;
uses
  Overlay, Dos, OpDos, OpTsr, OpCrt, OvrDemo1, OvrDemo2;
{$O OvrDemo1}
{$O OvrDemo2}

const
  TabKey = $080F; {ALT-TAB}
  Count : Word = 0;
var
  OvrName : string[64];

  procedure PopProc(var Regs : Registers);
    {-This is the actual routine that pops up when ALT-TAB is struck.}
  begin
    if not ReleaseEms then
      if not ExistFile(OvrName) then
        {error--don't call any overlayed routines!} ;

    {increment counter}
    Inc(Count);
    WriteLn('Inside popup. Count = ', Count, '.');

    {call routines in overlayed units}
    Write1;
    Write2;

    {disable the program after popping up 10 times}
    if Count >= 10 then
```

```
      if DisableTsr then
        WriteLn('Disabling OVRDEMO...');
  end;

begin
  {locate overlay file}
  if not ExistOnPath('OVRDEMO.OVR', OvrName) then begin
    WriteLn('Cannot find OVRDEMO.OVR!');
    Halt(1);
  end;

  {open overlay file}
  OvrInit(OvrName);
  if OvrResult <> 0 then begin
    Writeln('Overlay error: ', OvrResult);
    Halt(1);
  end;

  {move overlay into EMS if possible}
  OvrInitEms;
  case OvrResult of
    ovrIOError     : WriteLn('Overlay file I/O error.');
    ovrNoEMSDriver : WriteLn('EMS driver not installed.');
    ovrNoEMSMemory : WriteLn('Not enough EMS memory.');
  end;

  {if request for use of EMS succeeds, set ReleaseEms flag}
  ReleaseEms := (OvrResult = 0);

  {ALT Tab will activate this popup}
  if not DefinePop(TabKey, PopProc, Ptr(SSeg, SPtr), True) then
    WriteLn('Unable to create popup routine')
  else begin
    WriteLn('Pop up loaded, type ALT-TAB to activate.');
    PopupsOn;
    StayRes(ParagraphsToKeep, 0);
    {error if we get to here!}
    WriteLn('Unable to go resident');
  end;
end.
```

Although this program is weak on error checking, it does demonstrate two important points. First, you must be absolutely certain that your program has access to its overlay file when you pop up. The best way to ensure this is to obtain the full path name of the overlay file when your program starts, and use the ExistFile routine in OPDOS to check for its existence when you pop up. If the overlay file is not found, you should display an error message and exit from your popup--without calling any overlayed routines. Notice that checking for the overlay file is not necessary if EMS memory is being used to store the overlays.

Second, if you use the OvrInitEms routine to store the contents of your overlay file in EMS memory, you need to set the ReleaseEms variable in the OPTSR unit. This ensures that any EMS memory allocated by the overlay manager will be released when you disable your TSR. Notice that it is not necessary to save and restore the mapping context of the Expanded Memory Manager within your popup, since Turbo Pascal's overlay manager always preserves the mapping context.

Except for the matter of error handling, which you will want to improve upon in your own TSRs, that about covers it. If you follow the two basic guidelines described above, you should have little difficulty writing a solid TSR that uses Turbo Pascal's overlay system.

Modules

Have you ever accidentally tried to load the same TSR twice? Many TSRs aren't equipped to deal with that possibility, and the results of such an oversight usually aren't very pretty. OPTSR makes it easy for you to deal with this potential problem, however, by creating what we call a *standard TSR interface*. This facility allows any TSR that uses it not only to determine if it's already loaded, but also to communicate with all other TSRs (or modules) that use it. (More on that point in a moment.) Briefly, this is what you need to do to prevent your TSRs from being loaded twice: Assign each TSR a unique name, a string of say 5-10 characters. Before going resident, call ModuleInstalled to see if you're already resident:

```
const ModuleName : string[5] = 'MYTSR';
...
if ModuleInstalled(ModuleName) then begin
  WriteLn('Already installed. Aborting...');
  Halt;
end;
```

For this to work, of course, you must install the module before going resident the first time:

```
InstallModule(ModuleName, nil);
```

InstallModule adds your program (or module) to a linked list of records that store information about all programs in the system that use the standard TSR interface. From this point on, any program can find you by calling ModuleInstalled with your module name.

External Interfaces

Why would other programs want to find you? You may have noticed "nil" in the call to InstallModule above. The second parameter to InstallModule, CmdEntryRoutine, is the address of a procedure--called an "*external interface*"--that other programs can call to have you execute a particular function for them. Any program can locate your routine by means of this pointer, like so:

```
ExtIfcPtr := ModulePtrByName('MYTSR');
if ExtIfcPtr = nil then
  {MYTSR is not loaded}
else
  ExtIfcPtr := ExtIfcPtr^.CmdEntryPtr;
```

Your external interface procedure should be of the form:

```
procedure MyExternalInterface(BP : Word); interrupt;
var Regs : IntRegisters absolute BP;
begin
  with Regs do begin
    {--handle function calls--}
  end;
end;
```

Why is it an interrupt procedure? Three reasons. First, this allows it to be called easily using the EmulateInt routine in OPINT. Second, it allows you to pass parameters using the CPU registers, just as you do with Intr and MsDos. For example, let's say that MyExternalInterface does something special if Regs.AH := 1. You'd call it like this:

```
var Regs : IntRegisters;
...
Regs.AH := 1;
EmulateInt(Regs, ExtIfcPtr);
```

Third, by making the routine an interrupt procedure, we are relieved of the burden of switching data segments on entry to and exit from the routine. The compiler will generate the necessary code automatically.

The possible uses for this facility are unlimited. For an example of a TSR that has an external interface, see SMACS.PAS, and for an example of a program that calls it see PTIME.PAS.

Unloading a TSR

One very nice feature that many TSRs (including lots of well-known commercial ones) lack is a means for the user to unload it. Here again, OPTSR makes it easy for you to offer your users this option. From inside your popup, just call DisableTSR. If it returns True, your TSR will be disabled (unloaded from memory) as soon as you leave your popup. If it returns False, that means that it is very definitely unsafe to unload the TSR, so it won't be done. For a definition of "safe," see the documentation for SafeToDisable.

It is also possible to write a program that can unload itself from the DOS command line. If you wish to add this capability to your own programs, we recommend that you study the source code for SMACS, which provides a real-world example of how to do it. What follows is an abbreviated example of the basic technique:

```
procedure ExternalInterface(BP : Word); interrupt;
var
  Regs : IntRegisters absolute BP;
  SavePSP : Word;
begin
  {unload TSR if AH contains 'U'}
  if Char(Regs.AH) = 'U' then begin
    {save the current PSP} and switch back to our PSP}
    SavePSP := GetPSP;

    SetPSP(PrefixSeg);

    {try to unload the TSR--return success code in AL}
    AL := Ord(DisableTSR);

    {switch back to previous PSP}
    SetPSP(SavePSP);
  end;
end;

procedure UnloadFromCommandLine;
var
  Regs : IntRegisters;
  P : IfcPtr;
begin
  P := ModulePtrByName(OurModuleName);
  if (P <> nil) and (P^.CmdEntryPtr <> nil) then begin
    {restore all interrupt vectors}
    RestoreAllVectors;

    {make the function call}
    Regs.AH := Ord('U');
    EmulateInt(Regs, P^.CmdEntryPtr);
    if Regs.AL = 1 then
      {program unloaded successfully}
    else
      {unable to unload program} ;
  end;
end;
```

A few things about these two procedures require special comment.

First, although the two procedures are in the same program, there will be two copies of that program in memory at the time UnloadFromCommandLine is called. The first copy (A) has already gone resident; the second (B) has not. Our goal here is to have B tell A to disable itself if possible. That's why we have to use the external interface. Calling DisableTSR from within B doesn't accomplish anything.

Second, in ExternalInterface, you'll notice that there are calls to the inline macros GetPSP and SetPSP, both found in OPTSR. GetPSP returns the current Program Segment Prefix (or PSP), and SetPSP changes it. Each program has its own PSP, and hidden inside DOS is a variable that contains the PSP for the current program. GetPSP calls an undocumented DOS function that returns its value; SetPSP calls another documented function that changes it. It would take a small book to explain precisely why it is necessary to save, change, and restore the current PSP here, so please accept it as an article of faith: if you are going to call DisableTSR from your external interface, you have to call GetPSP and SetPSP. (The short explanation is that DisableTSR asks DOS to take back all the memory currently allocated to your program, and DOS can't do it if the PSP is wrong.)

Third, in UnloadFromCommandLine, notice that RestoreAllVectors is called before the indirect call to the external interface using EmulateInt. This is critical. The initialization code for OPTSR grabs several interrupt vectors, and they must be restored before copy B of your program asks copy A to disable itself. If you don't restore them, DisableTsr will think that it is not safe to unload and will therefore fail the call.

Everything else about the two routines is fairly self explanatory, with the possible exception of the business about loading Regs.AH with 'U'. This is completely arbitrary. If the sole function of the external interface is to unload the TSR, then AH does not need to be initialized by UnloadFromCommandLine or tested by ExternalInterface. If the external interface performs multiple functions, however, then you'll need to establish a set of function codes and use the registers to pass information back and forth. As indicated earlier, SMACS has an elaborate example of an external interface that you might want to study before writing your own.

Popups Everywhere

In the preceding paragraphs, it was assumed that you will be writing popups only for memory resident programs. This doesn't have to be the case. You can also use popups in "normal" programs. You must keep in mind, however, the problems of reentrancy discussed in the overview for the OPINT unit, since the popup will likely be called during the execution of your own program's code.

Getting Along with SideKick

SideKick can cause all sorts of problems for other TSRs, but OPTSR helps you out here in two ways.

First, during initialization, OPTSR checks to see if SideKick is in memory. If it is, it tries to patch the resident copy of SideKick in such a way as to allow other TSRs to be loaded safely after SideKick. The only known disadvantage to patching SideKick and then loading your TSR above it is that SideKick will no longer be able to keep itself from being loaded twice. We recognize that this is a sneaky trick, and if you'd prefer that your memory resident programs be mild mannered participants in the TSR wars, you can simply undefine the conditional compilation directive 'ThwartSideKick', found in OPDEFINE.INC, and recompile OPTSR.

Second, if SideKick's presence is detected, and OPTSR cannot find the proper patch point, it will set a global flag called 'SideKickLoaded' that you can check before going resident if you want to display an error message. Checking this flag is optional, but OPTSR will refuse to let you terminate and stay resident if SideKickLoaded is True.

OPTSR can also detect the presence of SideKickPlus, but it will leave the SideKickLoaded flag set to False if it is found, since it normally does not cause the same kinds of problems for other TSRs that SideKick does. You should almost always be able to safely load a TSR after SideKickPlus.

Playing It Safe

Before calling a popup that needs access to DOS services, the popup dispatcher takes several special precautions to ensure the safest possible environment for the popup to run in: 1) DOS's current BREAK level is saved, then "relaxed" to ensure that pressing <CtrlBreak> will not kill the popup. 2) Interrupts $1B (triggered by <CtrlBreak>) and $23 (triggered by <CtrlC> in certain situations) are both pointed to IRET instructions. 3) The current DTA (Disk Transfer Address) and PSP (Program Segment Prefix) are saved, and the ones that were current when the TSR was loaded are restored. 4) A DOS critical error handler is installed to allow the popup to detect and recover from these errors. When the popup returns control to the dispatcher, the system is restored to the state it was in before these steps were taken.

Technical Notes for Hackers

OPTSR always monitors the following interrupts: $05 (PrtSc), $08 (Clock tick), $09 (Keyboard), $13 (BIOS disk read/write), $16 (BIOS keyboard), $25 (DOS absolute disk read), $26 (DOS absolute disk write), and $28 (DOS multitasking). Interrupt $09 is monitored to watch for hotkeys; when a hotkey is pressed, the "pop ticker" is set to 36 (two seconds). The handlers for interrupts $08 and $28 together represent the "popup dispatcher": if the pop ticker is non-zero, they check to see if it is safe to pop up and, if so, invoke the appropriate popup. The other interrupts are merely "filtered," to ensure that popups will not be invoked while one of these interrupts is in progress. If the popup is being run in the OS/2 "compatibility box," three other interrupts are also filtered: $10 (BIOS video interrupt), $14 (BIOS communications), and $17 (BIOS printer). That's enough information to give you some idea of how popups are handled. If you want to know more, you'll need to study OPTSR.ASM.

Types

```
IfcPtr = ^IfcRecord;
```

Pointer to an IfcRecord.

```
IfcRecord = record
              NamePtr : ^String;
              Version : Word;
              CmdEntryPtr : Pointer;
              PrevIfc, NextIfc : IfcPtr;
            end;
```

A record containing data about a particular module. NamePtr points to the name of the module. Version is the version number of the interface. If not nil, CmdEntryPtr points to a routine that can be called by other programs. PrevIfc and NextIfc point to the previous and following entries in the linked list of modules.

```
PopupProc = procedure(var Regs : Registers);
```

Proper form for an OPTSR popup routine.

Constants

```
Int5Handle  = 1;
Int8Handle  = 2;
Int9Handle  = 3;
Int10Handle = 4;
Int13Handle = 5;
Int14Handle = 6;
Int16Handle = 7;
Int17Handle = 8;
Int25Handle = 9;
Int26Handle = 10;
Int28Handle = 11;
```

These are the ISR handles used by OPTSR's interrupt handlers. (Handle 12 is used by OPMACRO.)

```
MaxPopups = 8;
```

Maximum number of popup routines.

```
SideKickLoaded : Boolean = False;
```

Set to True by initialization code if SideKick is already loaded.

Variables

```
PopTickerPtr : ^Word;
```

Points to a variable hidden in OPTSR that indicates if we are currently trying to pop up. The value of PopTickerPtr^ tells the popup manager how many more times to attempt to activate the popup (attempts are made 18 times a second). This variable is intended primarily for internal use.

```
PopupInUse : array[1..MaxPopups] of Boolean;
```

These flags are used to keep track of which popups are currently active. They are intended primarily for internal use.

```
PopupsEnabledPtr : ^Boolean;
```

Points to a hidden variable that indicates if popups are enabled. This variable is intended primarily for internal use.

```
PopupToCallPtr : ^Byte;
```

Points to a hidden variable that contains the handle of the popup waiting to be called if PopTickerPtr^ is non-zero. This variable is intended primarily for internal use.

```
ReleaseEms : Boolean = False;
```

This variable needs to be set to True if a TSR is overlayed and the overlay manager is storing the overlay file in EMS.

```
ThisIfc : IfcRecord;
```

Record containing data about a particular module.

AddHotKey / ChangeHotKey

Declaration

```
function AddHotKey(PrimaryHotKey, SecondaryHotKey : Word) : Boolean;
```

Purpose

Add a second hotkey for the popup with the specified primary hotkey.

Description

AddHotKey provides a means of assigning multiple hotkeys to the same popup. PrimaryHotKey is the hotkey assigned to the popup when DefinePop was called. SecondaryHotKey is the additional hotkey to be assigned to it. Note that you can assign as many hotkeys as you want to the same popup; you are not limited to two.

AddHotKey returns False if it was unable to find the popup associated with PrimaryHotKey, or if the scan code portion of SecondaryHotKey conflicts with an existing hotkey. For example, you could not add <Ctrl><Tab> if <Alt><Tab> was the primary hotkey or if it was being used as a hotkey for another popup.

Example

```
const
  HotKey1 = $0829; {<Alt><~>}
  HotKey2 = $040F; {<Ctrl><Tab>}
...
  if DefinePop(HotKey1, PopupEntryPoint, Ptr(SSeg, SPtr), True) then
    if not AddHotKey(HotKey1, HotKey2) then
      {another hotkey based on <Tab> has been defined};
```

Correct. Either <Alt><~> or <Ctrl><Tab> will be able to activate this popup.

Declaration

```
function ChangeHotKey(OldHotKey, NewHotKey : Word) : Boolean;
```

Purpose

Change a popup's hotkey from OldHotKey to NewHotKey.

Description

This routine allows you to change the hotkey for a particular popup. The only restriction is that the scan code portion of the new hotkey must not conflict with the hotkey for any other popup. For example, you could change a hotkey from <Alt><Tab> to <Ctrl><Tab>, but you could not change it to <Ctrl><Space> if another popup had <Alt><Space> for a hotkey. ChangeHotKey returns False if it finds such a conflict.

Example

```
const
  OldHotKey = $0829;  {<Alt><~>}
var
  NewHotKey : Word;
...
  NewHotKey := $040F; {<Ctrl><Tab>}
  if not ChangeHotKey(OldHotKey, NewHotKey) then
    {hotkeys not the same, and new one already in use};
```

Changes the hotkey for the popup associated with <Alt><~> to <Ctrl><Tab>.

Declaration

```
function DefinePop(HotKey : Word; Routine : PopupProc;
                   StackPtr : Pointer; DosWait : Boolean) : Boolean;
```

Purpose

Defines a popup routine associated with a particular hotkey.

Description

Unlike DefinePopProc, this routine is used to define a popup that can be activated by HotKey. Routine is the name of the popup. StackPtr points to the end of the stack to be used by the popup. DosWait indicates whether or not the popup needs access to DOS services. DefinePop returns False if no more Popup handles are available.

Examples

```
const MyHotKey : Word = $0829; {Alt+ '~'}
var   MyStack : array[1..2048] of Byte;
...
if not DefinePop(MyHotKey, MyPopUp, @MyStack[2048], True) then {error};
```

Correct.

```
if not DefinePop(MyHotKey, MyPopUp, @MyStack[1], True) then {error} ;
```

Incorrect! '@MyStack[1]' is the address of the *start* of the alternate stack, not the *end* of it.

See Also

DeletePop

Declaration

```
function DefinePopProc(var Handle : Byte; Routine : PopupProc;
                       StackPtr : Pointer) : Boolean;
```

Purpose

Defines a routine that can be triggered by calling SetPopTicker.

Description

Unlike DefinePop, this routine is used to define a popup that can be activated by calling SetPopTicker. Routine is the name of the popup. StackPtr points to the end of the stack to be used by the popup. It is assumed that the popup needs access to DOS services. Handle is a number assigned to the popup by DefinePopProc; this number is needed in order to call SetPopTicker. DefinePopProc returns False if no more Popup handles are available.

Examples

```
var
  ClockHandle : Byte;
  ClockStack : array[1..1024] of Byte;
...
if not DefinePopProc(ClockHandle, ClockPop, @ClockStack[1024]) then {error};
```

Correct.

```
if not DefinePopProc(ClockHandle, ClockPop, @ClockStack[1]) then {error};
```

Incorrect! '@ClockStack[1]' is the address of the start of the alternate stack, not the end of it.

See Also

SetPopTicker DeletePopProc

DeletePop / DeletePopProc

Declaration

```
procedure DeletePop(HotKey : Word);
```

Purpose

Deletes the popup routine associated with HotKey.

Description

This routine lets you deactivate a popup defined with DefinePop. Its handle and hotkey(s) may then be used for other popups.

Examples

```
const MyHotKey = $0439;
...
DeletePop($0829);
DeletePop(MyHotKey);
```

See Also

DefinePop

Declaration

```
procedure DeletePopProc(Handle : Byte);
```

Purpose

Deletes a popup routine defined with DefinePopProc.

Description

This routine lets you deactivate a popup defined with DefinePopProc. Its handle may then be used for other popups.

Example

```
var   MyHandle : Byte;
      MyStack  : array[1..2048] of Byte;
...
if not DefinePopProc(MyHandle, MyPopup, @MyStack[2048]) then {error};
...
DeletePopProc(MyPopupHandle);
```

See Also

DefinePopProc

Declaration
```
function DisableTSR : Boolean;
```

Purpose
Disable TSR by restoring interrupt vectors and releasing memory.

Description
This routine lets you disable (or "unload") a TSR in many cases. Before disabling the program, it calls SafeToDisable to determine if it is safe to do so. If not, the function will return False. Otherwise, it will restore all the interrupt vectors that have been grabbed and release all of the program's memory to DOS. It *does not* actually halt the program. If DisableTSR returns True, your program should simply exit from the popup normally.

Warning: DisableTSR must not be called from a popup without access to DOS services.

Examples
```
if DisableTSR then Exit;
```
Correct.
```
if DisableTSR then Halt;
```
Incorrect! Never call Halt after your program has gone resident.

See Also
RemovePops SafeToDisable

Declaration
```
function DosBusyFlag : Byte;
```

Purpose
Returns current value of DOS busy flag.

Description
This routine, which is intended to be called from within a popup, indicates whether or not DOS was active at the time the popup was activated. If the value is 0, DOS was not active; otherwise, it was. Assuming that the popup has access to DOS services, the value returned should be non-zero only if the popup was activated while the user was at the DOS prompt.

Example
```
if (DosBusyFlag <> 0) then begin
  StuffString(^['TURBO.EXE'^M);
  Exit;
end;
```
If executed from inside a popup, this would stuff an <Esc>, the letters 'TURBO.EXE', and a carriage return into the keyboard buffer, then return to the DOS prompt.

Declaration
```
function DosCriticalFlag : Byte;
```

Purpose
Returns current value of DOS critical flag.

Description
This routine, intended to be called from within a popup, indicates whether a DOS critical error was pending when the popup was activated. If the value is 0, DOS was not critical; otherwise, it was. Note that under certain versions of DOS this value could conceivably be equal to the value returned by DosBusyFlag, and not have any special meaning. To see why, examine the InitTsrPtrs routine in OPTSR.ASM.

GetPSP / InitPops / InstallModule

Declaration
```
function GetPSP : Word;
```

Purpose
Returns current PSP.

Description
This function returns the PSP for the program that DOS thinks is currently active. It is intended to be used only in external interface routines. See the discussion of "Unloading a TSR" earlier in this chapter for details on what this routine does and how to use it.

See Also
SetPSP

Declaration
```
procedure InitPops;
```

Purpose
Initializes OPTSR's data structures.

Description
This procedure must be called before any other popup-related routines. Since it is called automatically when the program begins, you do not need to call it yourself, unless you are reinstalling popups after previously calling RemovePops.

Example
```
RemovePops;
{--something--}
InitPops;
```

See Also
RemovePops

Declaration
```
procedure InstallModule(var ModuleName : string; CmdEntryRoutine : Pointer);
```

Purpose
Installs the current program as a resident module that can be located and accessed by other programs.

Description
ModuleName is a string that uniquely identifies a TSR module. CmdEntryRoutine, if not nil, must point to the program's external interface, a pseudo-interrupt service routine that other programs can call using EmulateInt. InstallModule adds the module to the linked list of resident modules that use the standard TSR interface, thus enabling any program (including a duplicate copy of the current program) to find it by calling ModuleInstalled. The module can be detached from the list by calling UninstallModule.

Example
```
const MyTsrName : string[5] = 'MYTSR';
...
InstallModule(MyTsrName, nil);
```

See Also
EmulateInt (OPINT)
ModuleInstalled
UninstallModule

Declaration
```
function MaxParagraphsToKeep : Word;
```

Purpose
Maximum number of paragraphs to keep when going resident.

Description
This function is used internally to determine the maximum legal value of the ParasToKeep parameter passed to StayRes. If a program has been compiled with a max heap setting of 640K (the default), then MaxParagraphsToKeep will return the amount of memory that was available before the program was loaded. Hence, by subtracting ParagraphsToKeep (or the value to be passed to StayRes) from MaxParagraphsToKeep, you can calculate in advance approximately how much memory will be available after you go resident.

Example
```
KbLeft := LongInt(MaxParagraphsToKeep-ParagraphsToKeep) div 64;
WriteLn(KbLeft, 'K available memory');
StayRes(ParagraphsToKeep, 0);
```

See Also
ParagraphsToKeep

Declaration
```
function ModuleInstalled(var ModuleName : string) : Boolean;
```

Purpose
Returns True if ModuleName is installed.

Description
This function allows you to determine whether a particular module is installed. You can use it to prevent loading your own TSRs twice, as well as to check for the presence of another program (such as SMACS) that can be called on to execute special functions via its command entry pointer.

Examples
```
const MyModuleName : string[5] = 'MYTSR';
      OtherModuleName : string[10] = 'SUPER MACS';
...
if ModuleInstalled(MyModuleName) then begin
  WriteLn('Already loaded. Aborting...'); Halt(1);
end;
if not ModuleInstalled(OtherModuleName) then begin
  WriteLn('SMACS not loaded. Aborting...'); Halt(1);
end;
```

See Also
ModulePtrByName

ModulePtrByName / ParagraphsToKeep

Declaration

```
function ModulePtrByName(var ModuleName : string) : IfcPtr;
```

Purpose

Returns a pointer to the IfcRecord for the module named ModuleName or nil.

Description

Although superficially similar to ModuleInstalled, the main purpose for this routine is to allow you to locate the external interface for another module.

Example

```
const SmacsName : string[10] = 'SUPER MACS';
var P : IfcPtr;
  EntryPointer : Pointer;
begin
  P := ModulePtrByName(SmacsName);
  if P <> nil then EntryPointer := P^.CmdEntryPtr
  else EntryPointer := nil;
end;
```

This demonstrates how you would locate the external interface for SMACS.

See Also

ModuleInstalled

Declaration

```
function ParagraphsToKeep : Word;
```

Purpose

Returns # of paragraphs currently used for code, data, stack, and heap.

Description

This routine calculates the total amount of memory (in paragraphs) that your program is using. Its main use is to help you calculate the value to be passed as the ParasToKeep parameter to StayRes.

Examples

```
StayRes(ParagraphsToKeep, 0);
```

Go resident with no extra heap space.

```
StayRes(ParagraphsToKeep+1024, 0);
```

Go resident with 16K (1,024 paragraphs * 16 bytes/paragraph) of heap space available.

See Also

MaxParagraphsToKeep StayRes

Declaration
```
procedure PopupsOff;
```

Purpose
Turns popups off.

Description
This routine allows you to avoid conflicts among popups in programs that contain more than one. Popups can be reenabled with PopupsOn.

Declaration
```
procedure PopupsOn;
```

Purpose
Turns popups on.

Description
This routine enables popups. In TSRs, it should be called just prior to going resident. In conjunction with PopupsOff, it also allows you to avoid conflicts among popups in programs that contain more than one.

Examples
```
PopupsOn;
StayRes(ParagraphsToKeep, 0);
```
A standard sequence in a TSR: enable popups, then go resident.

```
{$F+}
procedure MyPopup(var Regs : Registers);
begin
  PopupsOff;
  ...
  PopupsOn;
end;
```
Here other popups are locked out until MyPopup ends.

Declaration
```
procedure RemovePops;
```

Purpose
Removes all popup routines from the system and releases all associated interrupt vectors.

Description
This routine is intended primarily for internal use. It is called automatically when a program ends. Note that this is not the same function performed by DisableTSR, which effectively halts a TSR.

See Also
DisableTSR InitPops

SafeToDisable / SetPopTicker / SetPSP

Declaration

```
function SafeToDisable : Boolean;
```

Purpose

Returns True if it is safe to disable the TSR.

Description

This routine allows you to determine whether or not it is safe to disable a TSR, without actually committing to doing so by calling DisableTSR. SafeToDisable first checks to be sure if the program has actually gone resident. If not, it returns False. It then checks to see if any other program has grabbed one of the interrupt vectors taken over by the TSR. If one has, then it's not safe to disable the TSR, and the function will return False. Otherwise it will return True.

See Also

DisableTSR

Declaration

```
procedure SetPopTicker(Handle : Byte; TimeOut : Word);
```

Purpose

Sets the pop ticker for the indicated procedure, clears interrupts, then returns.

Description

This tells the popup dispatcher to look for a good opportunity to call the popup associated with Handle. If TimeOut clock ticks go by without such an opportunity, the popup will not be called. (There are roughly 18.2 clock ticks per second.) *Very important:* No range checking is performed by SetPopTicker, so Handle absolutely must be a valid handle number (1..MaxPopups) returned by DefinePopProc.

Examples

```
SetPopTicker(MyPopupHandle, 36);
```

Correct. This means "try for the next two seconds to activate the popup associated with MyPopupHandle" (returned by DefinePopProc).

```
SetPopTicker(1, 36);
```

Incorrect! Don't use a constant as the Handle parameter, even if there's only one popup in your program.

See Also

DefinePopProc

Declaration

```
procedure SetPSP(PSP : Word);
```

Purpose

Sets current PSP.

Description

This function changes an internal DOS variable that is used to identify the currently active program. It is intended to be used only in external interface routines. See the discussion of "Unloading a TSR" earlier in this chapter for details on what this routine does and how to use it.

See Also

GetPSP

Declaration

```
procedure StayRes(ParasToKeep : Word; ExitCode : Byte);
```

Purpose

Terminate and stay resident, freeing up all our memory except ParasToKeep.

Description

This routine 1) releases all memory allocated to a program except ParasToKeep (number of paragraphs to keep, usually calculated by ParagraphsToKeep); 2) adjusts the heap manager's data structures so that memory can be allocated dynamically after going resident; 3) restores the interrupt vectors grabbed by the Turbo runtime library; and 4) calls the DOS Keep function ($31), passing it ParasToKeep and ExitCode (almost always 0) as parameters.

If StayRes is successful, the main program will end at the point of the call. StayRes will return control to the program only if it was unable to go resident, presumably because it was unable to release the program's memory.

Examples

```
StayRes(ParagraphsToKeep, 0);
```

Go resident with no extra heap space.

```
StayRes(ParagraphsToKeep+1024, 0);
```

Go resident with 16K (1,024 paragraphs * 16 bytes/paragraph) of heap space available.

See Also

MaxParagraphsToKeep
ParagraphsToKeep
PopupsOn

Declaration

```
procedure UninstallModule;
```

Purpose

Uninstalls the current module from the linked list of modules.

Description

This routine simply removes the current module from the linked list of modules. It is intended primarily for internal use (by DisableTSR, for example).

See Also

DisableTSR
InstallModule

D. Overview of Swappable TSRs

OPSWAP and OPSWAP1 implement a system with which you can write a TSR that uses as little as 6K of normal DOS memory, regardless of the actual size of the program. OPSWAP1 is very similar to OPTSR: it contains the routines that allow you to check whether the TSR is already installed, to define a popup routine, and to go resident. OPSWAP, however, is the key to swapping: it provides a small kernel of code that remains resident at all times, watching for hotkeys or other events. When OPSWAP detects a request to pop up, it first assures that system conditions are safe and then reads the remainder of the TSR's code and data from a secondary storage location in expanded (EMS) memory or disk. While doing this, of course, it is overwriting whatever program was loaded immediately after the TSR, so it also saves the current contents of these locations to secondary storage before overwriting them. After taking several other steps to ensure that the system is stable, OPSWAP transfers control to the popup routine of your program, which is written almost identically to that of an OPTSR popup. When your popup routine exits, OPSWAP reverses the steps and returns control to whatever program was interrupted.

Sounds simple, right? It *is* quite simple if you carefully follow the rules that we'll describe. As you can imagine, however, OPSWAP must play fast and loose with the rules of DOS programming. TSRs based on OPSWAP are quite reliable when they are running in a system whose behavior matches what OPSWAP expects. Unfortunately, today's PC environment includes a number of programs that are all pushing the limits of the system, and not all of them get along. Here are a few situations where an OPSWAP-based TSR is not appropriate:

- don't load an OPSWAP TSR before a network shell, a communications program, or any other program that receives real-time messages. When the TSR pops up, it must disable all of the programs loaded after, since it is overwriting their memory. In some cases, the side effects are relatively benign, like losing some characters in a communication session. In others, the system may hang. Loading the TSR after such programs is usually acceptable.
- don't load an OPSWAP TSR before multi-tasking system managers like DesqView, OmniView, and others. The reasoning is similar to the previous bullet, but the side effects in this case are never benign. Loading the TSR within a partition is fine, as long as the partition reserves enough memory for the full image of the TSR and not just its resident kernel.
- don't pop up an OPSWAP TSR while a program is active that uses a protected-mode DOS extender like Phar Lap's. DOS extenders play quite a few tricks themselves--remapping the 8259 interrupt controller, for example--as well as switching the processor into protected mode and, in the case of 386 systems, remapping memory. We've found that certain extenders, like the one from Rational Systems (used in Lotus 1-2-3 version 3.0), coexist nicely with swappable TSRs, while others, like the one from Phar Lap, don't get along at all.

TSRs that are triggered by events other than a hotkey are typically not appropriate for use with OPSWAP. The issue is not one of reliability, but rather one of practicality. Let's step back for a moment and see how this issue arises. Each time a swappable TSR pops up, OPSWAP must read its code and data from EMS or disk. How many bytes it reads depends on the complexity of the program, of course, but compared to OPTSR there is always an additional delay before your popup routine gets control. A large TSR, like the DESKPOP demonstration program, may take several seconds to swap in from disk. For hotkey activated TSRs, this delay is not usually a problem: in fact, OPSWAP provides a "swapping message" facility to soothe the impatient user during this period.

The most common example of a TSR that isn't triggered by a hotkey is one that watches the system timer. In this case an interrupt handler gets control at every timer tick, 18.2 times per second. Even if it were possible, you wouldn't want OPSWAP to be reading or writing your program's code and data four times (to save the underlying program, read in the TSR, write out the TSR, and restore the underlying program) at every one of those ticks.

It's possible to circumvent this limitation by keeping more of your program in memory than just OPSWAP's 6K kernel. Unfortunately, there's no good way to do this using just Pascal code; the additional resident portions must be written in assembly language. We've written such routines for POPHELP, the memory resident help database for Object Professional. POPHELP allows you to paste portions of the help system into an underlying editor. To do so, it provides an always-resident keyboard interrupt handler that feeds the keys on demand. If you'd like to see how this is done, take a look at the POPHELP source. Unfortunately, we can't generalize the technique to make it easy for everyone to do.

We've focused for a moment on the limitations of swappable TSRs, but we hope we haven't scared you off. In most situations, OPSWAP will allow you to create new memory-resident applications that would have been ungainly or downright impossible before now.

If you'd like to try using a swappable TSR before committing your own programs to the technology, we've provided several powerful ones for you to sample. DESKPOP, described in Chapter 1, is a complete programmer's desktop manager that would consume over 300K of memory if it weren't swappable. POPHELP, also described in Chapter 1, provides almost instant access to information about the routines in Object Professional. It would consume over 150K of memory without swapping. PSCREEN, described in Chapter 4, captures text screens and writes them to a disk file. While PSCREEN seems petite compared to the other two examples, its functions would probably not convince you to tie up 45K of memory for long. At 6K, however, you barely notice the cost. We're sure that the users of your TSRs will feel the same.

Writing Swappable TSR's

Subject to the restrictions we've just discussed, writing a swappable TSR is very similar to writing a normal one. In this section we'll tell how it's done, focusing primarily on the differences between using OPSWAP and using OPTSR. If you haven't already done so, you should read sections A through C of this chapter for additional background information.

In a swappable TSR, OPSWAP and OPSWAP1 take the place of OPINT and OPTSR (although the division of duties between the two units is not the same). You should not use either of the latter units in a swappable TSR; doing so will, at best, lead to wasted code and data and, at worst, cause conflicts that make your program crash.

OPSWAP and OPSWAP1 don't depend on any units from Object Professional other than OPINLINE. Hence, the swappable TSR capability can be used in programs written for Turbo Professional 5.0 (without the effort of converting to the new OOP interface tools) and it can be used with the Turbo Pascal 5.0 compiler as well as version 5.5.

Where you place OPSWAP and OPSWAP1 in the Uses statements of your program is the most important difference between using them and using OPTSR. The primary rule is that OPSWAP should be used *only* in the main program and should appear as the *last* unit in the Uses statement. By doing this, you force the compiler to link OPSWAP at the lowest segment in memory, coming after only the main program itself. The higher OPSWAP links, the larger the resident kernel will be. If you find that your TSR is retaining more memory than you think it should be, the first thing to do is to check the link order of the various units. To do so, have the

compiler generate a MAP file (using the /GS option for TPC, and Options/Linker/Map File/Segments for TURBO). The segment table at the top of the MAP file shows the link order of the units. If OPSWAP shows up anywhere below the second row of this table, there is a problem with the Uses statements.

Because the main program segment's code is always kept in memory, another requirement for minimizing the resident kernel size is to make the main program as small as possible.

OPSWAP1 may be used by any units as long as it doesn't appear after OPSWAP in the main program's Uses statement. OPSWAP1 provides the procedures that you would normally call in OPTSR, so you should use it wherever you need to call routines like InstallModule, DefinePop, and StayResSwap.

Putting these rules together leads to the program organization shown in the following small example. The entire main program file is as follows:

```
{$M 2048,0,655360}
{$S-,R-}
program BeepIt;
uses
  BeepMain,          {the main application code}
  OpSwap;            {resident portion of the swap manager}
begin
  InitializeBeepIt;
end.
```

BEEPMAIN is a unit that takes the place of what you would normally put in the main program. By moving the code to BEEPMAIN, the OPSWAP system can swap it out of memory. InitializeBeepIt is the only procedure interfaced by BEEPMAIN and BEEPIT's main block calls this procedure to start the action. The other features of this program, particularly the stack and heap sizes in the $M directive, are treated exactly the same as for OPTSR.

Now let's look at BEEPMAIN.PAS:

```
{$S-,R-,V-,I-}
unit BeepMain;
  {-Sample swappable TSR main unit}
interface
uses
  OpInline,          {assorted inline macros}
  OpString,          {assorted string routines}
  OpCrt,             {video and keyboard services}
  OpSwap1;           {non-resident portion of the swappable TSR manager}

procedure InitializeBeepIt;
  {-Initialize the TSR: check for install, init popups, and go resident}

implementation

const
  HotKey    = $0844;                          {<Alt><F10>}
  OurName   : String[9]  = 'BEEPIT1.0';       {our Module's name}
  SwapFile1 : String[15] = 'C:\BEEPSWP1.$$$'; {swap file 1 name}
  SwapFile2 : String[15] = 'C:\BEEPSWP2.$$$'; {swap file 2 name}
```

```
{$F+}
procedure MainPop;
  {-This procedure is called when the hotkey is pressed}
var
  I : Word;
begin
  for I := 1 to 100 do begin
    Sound(I*10);
    Delay(1);
  end;
  NoSound;
end;
{$F-}


procedure InstallCheck;
  {-Abort if this program is already installed}
begin
  if ModuleInstalled(OurName) then begin
    WriteLn('BEEPIT already installed');
    Halt;
  end;
end;

procedure InitOurPopups;
  {-Define the popup routine, enable popups,and install this module}
begin
  if not DefinePop(HotKey, MainPop, Ptr(SSeg, SPtr)) then begin
    WriteLn('Error defining popup procedure');
    Halt;
  end;
  PopupsOn;                                  {enable popups}
  InstallModule(OurName, DisablePopup);      {install the module}
end;

procedure TryToUnload;
  {-If BeepIt already loaded, attempt to unload it and report results}
var
  IfcP : IfcPtr;
begin
  IfcP := ModulePtrByName(OurName);
  if IfcP = nil then begin
    WriteLn('BEEPIT not loaded, so it cannot be unloaded!');
    Halt;
  end;

  {restore all vectors taken over by OPSWAP's initialization code}
  RestoreAllVectors;

  {call the CmdEntryPtr for the resident copy of BeepIt}
  IfcP^.CmdEntryPtr;
  if LongInt(IfcP^.UserData) = 1 then
    WriteLn('BEEPIT unloaded')
  else
    WriteLn('Unable to unload BEEPIT');
end;
```

```
procedure CheckCommandLine;
  {-Check for command line parameters}
var
  S : String;
begin
  if ParamCount > 0 then begin
    S := StUpcase(ParamStr(1));
    if S = '/U'  then
      TryToUnload
    else
      WriteLn('Unknown option ', S);
    Halt;
  end;

end;

procedure InitializeBeepIt;
  {-Initialize the TSR: check for install, init popups, and go resident}
begin
  CheckCommandLine;
  InstallCheck;
  InitOurPopups;
  WriteLn('Going resident, <Alt><F10> to popup...');
  StayResSwap(ParagraphsToKeep, 0, SwapFile1, SwapFile2, True);
  WriteLn('Error going resident');
end;

end.
```

BEEPMAIN shows all of the major steps necessary for a full-featured, hotkey-based, swappable TSR. It can unload itself (based on the /U command line option), detect previous installation and report the error, and install itself to pop up. Let's go through this unit in some detail to see how all of these are done.

First, note BEEPMAIN's Uses statement. It must use OPSWAP1, since most of the functions we're going to describe are located in that unit. Just as important, it does *not* use OPSWAP, for the reasons we've already mentioned. Any other units that appear in BEEPMAIN's Uses statement are incidental to the function of the popup.

BEEPMAIN interfaces just the procedure InitializeBeepIt. There's no point in it interfacing anything else, since the main program won't use it anyway.

InitializeBeepIt is responsible for checking the DOS command line and installing or uninstalling the popup. To do so, it calls CheckCommandLine where, if '/U' is found, TryToUnload attempts to remove the TSR from memory. This routine calls ModulePtrByName to see whether the TSR has been loaded before. Just as with OPTSR, ModulePtrByName returns a non-nil pointer (to an external interface record) if the TSR has been loaded. In BEEPIT's case, the CmdEntryPtr field of that record has been set up to unload the TSR. The statement 'IfcP^.CmdEntryPtr' makes a far call into the previously loaded copy of BEEPIT, effectively telling it to unload itself. Unlike OPTSR, OPSWAP's external interface routines don't take a set of registers as a parameter. Instead, they can communicate via the UserData field of the record. We'll return to this issue later; note for now that UnloadRequested checks whether the unload request was successful by looking at the value of the UserData field.

If an unload was not requested, InitializeBeepIt continues by checking for a previous installation of the TSR. It calls InstallCheck, which uses OPSWAP1's ModuleInstalled function exactly the same as the version in OPTSR.

InitOurPopups defines the popup--its hotkey, the user-written routine that will gain control, and the stack it will use--by calling DefinePop. This routine is almost identical to the version in OPTSR. The first difference is that OPSWAP1's DefinePop does not take a DosWait parameter. The popup routines of swappable TSRs must always wait for DOS to be available because DOS services are required to swap the TSR into position.

There is another important difference in OPSWAP1's DefinePop: the popup procedure itself (MainPop in this example) is defined differently than for OPTSR. In OPTSR, the popup procedure is declared like this:

```
{$F+}
procedure OpTsrPopup(var Regs : Registers);
```

For OPSWAP, popup procedures *do not* take a Regs parameter:

```
{$F+}
procedure OpSwapPopup;
```

It's a rare TSR that needs to refer to the registers of the program that it interrupted. If your application needs to do so, then a swappable TSR is not the way to go about it. Except for this difference, the popup routine of a swappable TSR can do all the same things that one written for OPTSR can do.

The rest of InitOurPopups looks like the corresponding OPTSR code, but it again has a subtle yet important difference. The second parameter to InstallModule specifies the external interface procedure for the TSR. In OPTSR, the external interface procedure is an interrupt procedure:

```
procedure OpTsrInterfaceProc(BP : Word); interrupt;
```

For OPSWAP, the interface is simply a FAR procedure without parameters:

```
{$F+}
procedure OpSwapInterfaceProc;
```

DisablePopup is a predefined external interface procedure that OPSWAP1 supplies for the single purpose of unloading a TSR.

We've almost reached the end of BEEPMAIN now. It has one more step--going resident. OPSWAP1 provides a specialized version of the related routine (StayRes) in OPTSR. StayResSwap specifies the number of paragraphs for the popup to keep just like OPTSR's version. Note that this is the number of paragraphs the TSR will have once it has swapped in, not the size of the resident kernel. You can't control the size of the kernel except by the linking mechanisms already described. StayResSwap takes three additional parameters. Two of them specify the names of the swap files that OPSWAP will use when swapping to disk. The last parameter, a Boolean, specifies whether OPSWAP should use swapping at all. If this parameter is False, OPSWAP drops back into a mode that works almost like OPTSR--the TSR keeps all the memory specified by ParasToKeep and never swaps out at all.

Once BEEPIT is installed, you can press the hotkey (<Alt><F10>) and it will pop up, make a sound, and pop back down. If you want to uninstall it, enter BEEPIT /U at the command line.

What we've discussed in this section is enough to let you write a swappable TSR. The following sections provide additional information that will help you to understand and tune your swappable TSRs.

Swapping To Disk or EMS

OPSWAP uses either disk files or EMS memory for secondary storage of the swapped-out information. It will use EMS when there is sufficient free space available there. EMS is preferred, of course, because it's faster than disk storage and not subject to issues such as the user deleting the swap files or removing the diskette containing them.

In some cases, you may want to reserve a system's EMS memory for other programs besides the TSR. To keep a swappable TSR from using EMS, set the OPSWAP1 global variable SwapUseEMS to False before calling StayResSwap.

When a swappable TSR is not using EMS, you must be careful about choosing the location of the swap files. As you've seen, StayResSwap requires that you name two swap files. OPSWAP uses one of them to store an image of the program that it's overwriting and the other for the contents of the TSR itself. You don't need to be concerned about which one is which. You do need to be concerned with pathnames of the files. Here are a few rules and recommendations:

- Each of the two files uses approximately the same number of bytes as the value specified for the first parameter to StayResSwap (ParasToKeep), after multiplying it by 16. The drive must have space to hold both files simultaneously. (The two files could go on different drives if required.)
- The pathnames must completely specify a drive, directory, and filename. OPSWAP opens and closes the files at every popup and must be able to find the same ones even if the user has changed to a different drive or directory.
- The files should not be located on a removable disk. OPSWAP does not enforce this recommendation and you are welcome to ignore it. Be warned that if OPSWAP cannot find a swap file when it needs it, OPSWAP will be forced to present a prompt that only allows the user to reboot the machine. To detect a removable disk, use the GetDiskClass function from OPDOS. (The PSCREEN example program shows how.)
- A RAM disk is the preferred form of swap file storage. In fact, a RAM disk located in extended memory is the only way for OPSWAP to utilize that form of memory.
- To provide acceptable flexibility, your TSR should allow the user to specify a preferred location for the swap files.

OPSWAP marks the swap files with the Hidden and System attributes to make it harder for someone to delete them inadvertently. OPSWAP automatically deletes the swap files when the TSR is unloaded. If the machine is turned off or rebooted without unloading the swappable TSR, the swap files will remain in place. However, OPSWAP will overwrite the same files (as long as the same swap pathnames are specified) the next time the TSR is loaded.

When OPSWAP is using EMS, it keeps the memory allocated until the TSR is unloaded. A swappable TSR can peacefully coexist with other programs that use EMS, even when it pops up over a program that is using EMS.

Swapping Messages

When swapping to a slow disk, a TSR written with OPSWAP may take longer to pop up than the user expects. OPSWAP therefore provides the capability to display a message while swapping in and out. OPSWAP1 interfaces several routines that allow you to customize the behavior of the swap messages.

By default the messages are enabled. They appear on row 25 of the display in reverse video (attribute $70). When the TSR is popping up, OPSWAP saves the current contents of the entire row, writes the message 'Swapping in...' to column 1, and clears all 80 columns of the row in the same attribute. After the swap is complete, OPSWAP restores the previous contents of the message row. When the TSR is popping down, OPSWAP performs the same steps using the message 'Swapping out...'. OPSWAP uses BIOS video services (interrupt $10) for all of these activities.

When the swapping time is short, as is the case for both small TSRs and larger ones swapping to EMS or a RAM disk, the messages are more of a distraction than a boon. In this case, you can call SetSwapMsgOn(False) to disable the messages. It's usually a good idea to supply a command line option that allows the user to decide whether swap messages should appear.

You can adjust the video attribute for the message by calling SetSwapMsgAttr and the row where it appears by calling SetSwapMsgRow. The latter procedure offers a special capability that allows you to specify the last row of the screen even when 43 or 50 line mode is in use. If you ask for row 255 ($FF), OPSWAP will interrogate the BIOS to determine the last row number. This capability is available only on MCGA, EGA, and VGA video adapters, however, and it's your responsibility to use it correctly. The following sequence of code uses OPCRT's CurrentDisplay variable to force the message row safely to the bottom of the screen:

```
case CurrentDisplay of
  MCGA, EGA, VGA : SetSwapMsgRow(255);
else
  SetSwapMsgRow(25);
end;
```

You cannot change the text of the swapping messages without modifying the source code for OPSWAP.ASM. If you need to do so, modify the SwapInMsg and SwapOutMsg constants there and reassemble it.

The PSCREEN example program shows how to combine the various swap message options. See PSMAIN.PAS.

External Interfaces

Like a TSR written with OPTSR, a swappable one provides external interface capabilities--a way for other programs to find the TSR and access its internal functions. Although the capabilities are similar on the surface, OPSWAP's external interface must be implemented quite differently because most of the TSR is not in memory when an external interface request occurs. In this section, we'll review the use of the external interface and point out the differences in using one with OPSWAP.

An interface is set up just the same as for OPTSR. You declare a module name for your TSR and optionally provide a "command entry routine," which is the routine that other programs will be able to call:

```
const
  OurName    : String[9]  = 'BEEPIT1.0';        {our Module's name}
...
InstallModule(OurName, EntryRoutine);
```

The string (OurName) allows other programs to find the TSR by name; the other parameter to InstallModule is the command entry routine itself.

The most common need for an external interface is simply to detect whether the TSR has been installed previously--in order to avoid installing it a second time. OPSWAP1 provides the ModuleInstalled method specifically for this purpose:

```
if ModuleInstalled(OurName) then begin
  WriteLn('BEEPIT already installed');
  Halt;
end;
```

If you just want to check for previous installation, you don't need a command entry routine at all. OPSWAP interfaces an identifier, NoExternalIfc, which may be passed as the second parameter to InstallModule in this case:

```
InstallModule(OurName, NoExternalIfc);
```

If you want to be able to unload your TSR from the DOS command line, you'll make actual use of a command entry routine. OPSWAP1 provides a routine specifically for this purpose. Just pass DisablePopup as the external interface routine:

```
InstallModule(OurName, DisablePopup);
```

The BEEPIT example already showed you how to access a command entry routine. Here we'll give a slightly more thorough example that also demonstrates how to control the swapping messages while the TSR is being disabled.

```
procedure TryToUnload;
  {-Attempt to remove TSR from memory}
var
  IfcP : IfcPtr;
begin
  IfcP := ModulePtrByName(OurName);
  if IfcP = nil then begin
    WriteLn(OurName, ' not loaded, so it cannot be unloaded!');
    Halt;
  end;
  if @IfcP^.CmdEntryPtr = nil then begin
    WriteLn(OurName, ' has no command entry routine!');
    Halt;
  end;

  {restore all vectors taken over by OPSWAP's initialization code}
  RestoreAllVectors;

  SaveMsgOn := IfcP^.CSDataPtr^.SwapMsgOn;    {save state of swap messages}
  IfcP^.CSDataPtr^.SwapMsgOn := False;        {disable swap messages}
  IfcP^.CmdEntryPtr;                          {call the CmdEntryPtr}
  if LongInt(IfcP^.UserData) = 1 then         {check status of unload attempt}
    WriteLn(OurName, ' unloaded')
  else begin
    WriteLn('Unable to unload ', OurName);
    IfcP^.CSDataPtr^.SwapMsgOn := SaveMsgOn;  {restore state of swap messages}
  end;
end;
```

ModulePtrByName returns a pointer to the external interface record of the TSR, or nil if none is found. The comparison of @IfcP^.CmdEntryPtr to nil assures that a command entry routine was correctly installed. RestoreAllVectors gives back the interrupt vectors that OPSWAP grabs so that the already-resident copy of the TSR will conclude that it's safe to unload.

The next sequence of code demonstrates the differences between OPSWAP and OPTSR. In OPSWAP's case, much of the data needed by the TSR manager is hidden within OPSWAP's code segment, and it may be referred to by using the CSDataPtr field of the interface record. Unlike OPTSR, OPSWAP can't keep its data in the TSR's data segment because the data segment is overwritten by another application during most of the time it's needed. Most of the information that CSDataPtr points to is for the internal use of the swap manager. However, there are a few fields that you may want to refer to. One of them, SwapMsgOn, is shown here. We save its current value, disable swapping messages temporarily, call the CmdEntryPtr routine to unload the TSR, and then restore the original state of SwapMsgOn. This sequence keeps the status message from flashing while the TSR is unloaded. It also makes clear the fact that OPSWAP swaps the entire TSR into memory before it transfers control to the actual command entry routine.

Note that the example we've given is intended to allow you to disable a TSR from a second copy of the program (or another application altogether) that has been run at the DOS command line. If you'd like to unload the TSR while it is popped up, just call the DisableTSR function provided by OPSWAP1:

```
{$F+}
procedure MyPopup;
begin
  ...
  if UnloadWasRequested then
    if DisableTsr then
      WriteLn('TSR unloaded')
    else
      WriteLn('Unsafe to unload at this time');
end;
{$F-}
```

The external interface may be used for purposes other than unloading the TSR--that's why we've designed InstallModule to take a parameter that specifies the name of a particular command entry routine. The external interface might allow other programs to make calls to the TSR, using it as kind of a memory-resident runtime library. Or another program might just access some data that the user has entered while running the TSR.

When considering such possibilities, keep in mind that OPSWAP must swap the entire TSR into memory and out again whenever you call the CmdEntryPtr routine. This is a great advantage from the CmdEntryPtr routine's perspective: once it gains control, it can use Pascal code to perform any normal program actions. However, the time required for the swap may rule out the use of the external interface for some applications.

If, however, the need for the external interface is just to access some of the TSR's data, there's another possibility. If the TSR stores the data in a location that always remains resident, another application can reference or change that data without requiring a swap. Choosing an appropriate location is tricky, since the TSR's data segment, stack, and heap are likely to be overwritten by the application that cares to read them. That leaves a few other possibilities:

- memory below the TSR, like unused interrupt vectors or the BIOS interapplications communication area (ICA)
- the TSR's prefix segment
- space reserved in the TSR's resident code segment

The first option isn't recommended, since the amount of memory space is quite limited and other applications may be competing for the same space. The second option is the easiest to

use and provides a convenient 128 bytes of space where the TSR's command line was originally located. The third option offers the largest possible area to share, but it directly increases the size of the TSR's resident kernel. To reserve such an area requires that you declare space in the code segment of an assembly language file linked into the main program or that you create a "fake procedure" with Borland's BINOBJ utility and link it into the main program.

Let's assume that you've chosen one of these approaches and stored the address of the shared memory area in a pointer named P. To initialize P for the BIOS ICA, for example, you'd say:

```
P := Ptr($40, $F0);
```

To initialize P for the TSR's prefix segment, you'd say:

```
P := Ptr(PrefixSeg, $80);
```

And to initialize P for space in the TSR's resident code segment, you'd say:

```
P := @MyDummyProcedure;
```

Now, how do you go about accessing this data? First, before the TSR goes resident, it must store the pointer within the external interface record so that other programs can find it. It does so with the following statement:

```
CSSwapData^.ThisIFC.UserData := P;
```

CSSwapData is a global variable interfaced by OPSWAP1.

When the TSR is popped up, it can read or write to the shared area simply by dereferencing the pointer P.

For another program to access the data, it uses the following sequence:

```
var
  IfcP : IfcPtr;
  P : OurDataPointer;
begin
  IfcP := ModulePtrByName(OurName);
  if IfcP = nil then begin
    {The TSR isn't loaded now}
    Halt;
  end;
  P := IfcP^.UserData;
  {Dereference the pointer P}
  ...
```

To reiterate: this approach is useful when the TSR and another application need to share data. It avoids any delay associated with swapping the TSR into and out of memory.

When an application actually needs to activate the TSR by calling its command entry routine, swapping is unavoidable. In this case, the UserData field of the interface record may be used to pass parameters into the command entry routine and to return simple results. Here's an example of a multi-purpose command entry routine that disables the TSR under one condition and writes a message in another:

```
{$F+}
procedure EntryRoutine;
var
  ParamCode : Byte;
begin
  {Get the parameter code from the interface record}
  ParamCode := LongInt(CSSwapData^.ThisIFC.UserData);
  case ParamCode of
    0 : {Disable the TSR}
      ParamCode := Byte(DisableTSR);
    1 : {Write a message}
      begin
        WriteLn('Entered the external interface with code 1');
        ParamCode := 1; {Return a success code}
      end;
  end;
  {Return a status result in the interface record}
  LongInt(CSSwapData^.ThisIFC.UserData) := ParamCode;
end;

{$F-}
```

Another program would use a procedure like the following to call the external interface:

```
procedure CallExternalInterface(ParamCode : Byte);
var
  IfcP : IfcPtr;
  P : OurDataPointer;
begin
  IfcP := ModulePtrByName(OurName);
  if IfcP = nil then begin
    {The TSR isn't loaded now}
    Halt;
  end;
  if @IfcP^.CmdEntryPtr = nil then begin
    {The TSR has no command entry routine}
    Halt;
  end;

  {restore all vectors taken over by OPSWAP's initialization code}
  RestoreAllVectors;

  LongInt(IfcP^.UserData) := ParamCode;        {store the parameter code}
  SaveMsgOn := IfcP^.CSDataPtr^.SwapMsgOn;     {save state of swap messages}
  IfcP^.CSDataPtr^.SwapMsgOn := False;         {disable swap messages}
  IfcP^.CmdEntryPtr;                           {call the CmdEntryPtr}
  ParamCode := LongInt(IfcP^.UserData);        {get the returned parameter}
  IfcP^.CSDataPtr^.SwapMsgOn := SaveMsgOn;     {restore state of swap messages}
  ...                                          {check the returned result}
end;
```

External interfaces are an advanced and somewhat confusing topic. Once mastered, however, they open up interesting possibilities for new applications.

Differences Between OPTSR and OPSWAP

We've already noted many of the issues you'll need to know when writing a TSR for OPSWAP rather than OPTSR. In this section, we'll summarize those differences and mention a few new ones.

The popup routine for OPSWAP is declared as a FAR procedure without parameters:

```
{$F+}
procedure OpSwapPopup;
```

A command entry routine for an OPSWAP external interface is also declared as a FAR procedure without parameters:

```
{$F+}
procedure OpSwapInterfaceProc;
```

An OPSWAP TSR calls StayResSwap rather than StayRes to go resident. StayResSwap takes three extra parameters. In addition to ParasToKeep and ExitCode, StayResSwap also requires two pathnames and a Boolean. The pathnames are used for the swap files. (They are ignored if OPSWAP uses EMS for secondary storage.) If the Boolean, UseSwapping, is False, the application goes resident in the traditional manner (like StayRes in OPTSR).

OPSWAP1 provides a new routine called WillSwapUseEms. This routine may be called before going resident to predict whether OPSWAP will use EMS for secondary storage. The results may be used to determine whether to disable the swapping message automatically, and to write a status message for the user.

OPSWAP1 does not provide a routine that is equivalent to OPTSR's DefinePopProc. This is consistent with the fact that almost all swappable TSRs will be triggered by a hotkey (assigned with DefinePop) rather than by another event like a timer tick. OPSWAP does provide a SetPopTicker routine. See the next section for more information about non-hotkey triggered swappable TSRs.

Breaking the Rules

While listing the rules for writing TSRs with OPSWAP we said that OPSWAP must be the last unit used in the Uses list of the main program file. There are times when it may be necessary to break this rule. Any unit listed after OPSWAP in the Uses list of the main program, as well as the code of the main program itself, remains permanently resident with OPSWAP's kernel. You can take advantage of this fact if you're a) an assembly language programmer and b) very careful.

The goal is for the permanently resident portion of the program to do more than just watch for hotkeys. Perhaps it might continuously update an on-screen clock, or wait for a particular time of day to trigger an event, or playback keystrokes for a non-resident application. You've already seen one such example of this behavior: the memory-resident help system POPHELP is a swappable TSR that can paste help topics into an underlying editor.

Achieving this goal is difficult because the permanently resident portion of the TSR lacks several important components that a Pascal program normally counts on. Specifically, it is missing most of the program's code (including the SYSTEM unit), all of the global data, its stack, and its heap. All it has left is the code in the main program and part of OPSWAP.

The only OPSWAP facility that your code is likely to use in this situation is the SetPopTicker routine. Call SetPopTicker to request that the entire TSR be swapped in and the CmdEntryPtr routine called. At that time the TSR can work like a real Pascal program. For an alarm clock style TSR, you can write a simple timer interrupt handler that checks for the scheduled time and calls SetPopTicker when the time comes.

We'd like to explain more, but this manual is long enough without including an assembly language tutorial. If you'd like to see how to break the rules and survive, see the source code for POPHELP and also for the ALARM program we've provided on the BONUS diskette.

E. OPSWAP1: Swappable TSRs, Part 1

With the possible exception of SetPopTicker, OPSWAP1 provides all of the variables and routines that a swappable TSR will ever need to use. Additionally, many of the routines that OPSWAP1 provides are declared and used just like ones of the same name in OPTSR. Such routines are listed at the end of the following declarations. Refer to OPTSR for more information.

Constants

```
EmsAllocated : Boolean = False;
```

Set to True if EMS memory has been allocated by the popup.

```
FileAllocated : Boolean = False;
```

Set to True if disk files have been created by the popup.

```
MaxPopups = 8;
```

Maximum number of popups that may be defined.

```
MaxScanCode = $58;
```

The highest scan code OPSWAP can use. This value may not be changed. This is the scan code of the <F12> key.

```
ReleaseEms : Boolean = False;
```

If True, a call to DisableTSR will release the EMS handle used by the Turbo Pascal overlay manager. OPSWAP has no way of knowing whether the overlay manager is present and using EMS, so it is your responsibility to set this to True in that case.

```
SideKickLoaded : Boolean = False;
```

Set to True by initialization code if SideKick is already loaded.

```
SwapFileMaxLen  = 67;
```

The maximum length of a swap file name.

```
SwapUseEMS : Boolean = True;
```

If False, OPSWAP will not use EMS for swapping even if sufficient EMS is present.

Types

```
CSSwapDataPtr = ^CSSwapDataType;
CSSwapDataType =
  record
    ...
    SwapMsgOn : Boolean;
    SwapMsgAttr, SwapMsgRow  : Byte;
    ThisIfc : IFCRecord;
    ModuleName : Str20;
    IfcInstalled : Boolean;
    ...
    SwapEnabled : Boolean;
    ...
  end;
```

The record definition that defines the data stored in the code segment of OPSWAP. This record structure *must not* be changed. This record is primarily for internal use, but needs to be interfaced so OPSWAP.PAS can access it. Some of the fields may be useful to applications programmers, and those are shown here. SwapEnabled is set to True if the TSR is using swapping. See the overview of swappable TSRs (section D) for more information.

```
IfcPtr = ^IfcRecord;
IfcRecord =
  record
    NamePtr : ^string;
    Version : Word;
    CmdEntryPtr : PopupProc;
    PrevIfc, NextIfc : IfcPtr;
    CSDataPtr : CSSwapDataPtr;
    UserData  : Pointer;
  end;
```

The module interface record and pointer type. This data type *must not* be changed. The last two fields, CSDataPtr and UserData, are unique to this unit. CSDataPtr is a pointer to the code segment relative data within OPSWAP. UserData is a pointer field designed to pass data to and from the command entry procedure.

```
IntRegisters =
  record
    case Byte of
      1 : (BP, ES, DS, DI, SI, DX, CX, BX, AX, IP, CS, Flags : Word);
      2 : (Dummy : Dummy5; DL, DH, CL, CH, BL, BH, AL, AH : Byte);
  end;
```

A record whose structure mirrors the contents of the stack frame in an interrupt procedure.

```
PopupProc = procedure;
```

The procedure type used for popup procedures and external interface procedures.

Procedures and Functions

The following procedures and functions are identical to the routines of the same name in OPINT or OPTSR:

```
function AddHotKey(PrimaryHotKey, SecondaryHotKey : Word) : Boolean;
  {-Adds a second hot key for the popup with the specified primary hot key}

function AllocateStack(var P : Pointer; SizeInBytes : Word) : Boolean;
  {-Allocates a stack of size SizeInBytes}

function ChangeHotKey(OldHotKey, NewHotKey : Word) : Boolean;
  {-Changes a popup's hotkey from OldHotKey to NewHotKey}

procedure DeallocateStack(P : Pointer);
  {-Frees an allocated stack}

function DisableTSR : Boolean;
  {-Disables TSR by restoring interrupt vectors and releasing memory. This does
    not halt the program. Returns False if it's not safe.}

function DosBusyFlag : Byte;
  {-Returns current value of DOS busy flag}

procedure EmulateInt(var Regs : IntRegisters; IntAddr : Pointer);
  {-Emulates an interrupt by filling the CPU registers with the values in
    Regs, clearing interrupts, pushing the flags, and calling far to IntAddr}

procedure InterruptsOff;
  {-Turns interrupts off}

procedure InterruptsOn;
  {-Turns interrupts on}

function ModuleInstalled(var ModuleName : string) : Boolean;
  {-Returns True if ModuleName is installed}

function ModulePtrByName(var ModuleName : string) : IfcPtr;
  {-Returns a pointer to the IfcRecord for the module named ModuleName or Nil}
```

```
function ParagraphsToKeep : Word;
  {-Returns number of paragraphs currently used for code, data, stack, and
    heap}
procedure PopupsOff;
  {-Turns Popups off}.
procedure PopupsOn;
  {-Turns Popups on}
procedure RestoreAllVectors;
  {-Restores all captured interrupt vectors}
function SafeToDisable : Boolean;
  {-Returns True if it is safe to disable the TSR}
procedure UninstallModule;
  {-Removes the module from the linked list of modules}
```

DefinePop

Declaration

```
function DefinePop(HotKey : Word; Routine : PopupProc;
                   StackPtr : Pointer) : Boolean;
```

Purpose

Defines a popup routine associated with a particular hotkey.

Description

This routine is similar to the DefinePop routine in OPTSR. The key difference is that the PopupProc in an OPSWAP TSR is a FAR procedure without parameters. As with OPTSR, StackPtr must be a pointer to the *top* of the stack for this popup to use. See the OPTSR section for more information on stacks.

Since OPSWAP itself needs DOS services to swap the program in, DOS services are always available to popups in swappable TSRs.

Example

```
{$F+}
procedure PopupEntryPoint;
begin
  ...
end;
{$F-}
...
if not DefinePop(OurHotKey, PopupEntryPoint, Ptr(SSeg, SPtr)) then
  Abort('Cannot define popup procedure');
```

See Also

DefinePop (OPTSR)

Declaration

```
procedure DisablePopup;
```

Purpose

An external interface procedure (CmdEntryPtr) that attempts to disable the popup and reports the results in the UserData field of the IfcRecord.

Description

This routine is a shell around DisableTSR that allows it to be used in the CmdEntryPtr parameter of InstallModule. It provides an easy method of allowing a TSR to be disabled by another program (or by the same program when run again from the command line).

When DisablePopup is specified as the CmdEntryPtr, calling the CmdEntryPtr will instruct the TSR to unload itself. The UserData field of the IfcRecord for the module will be set to 1 if the TSR was successfully unloaded, otherwise it will be zero. The UserData field is of type Pointer, so a typecast to LongInt is required to check for 0 or 1.

Example

```
InstallModule(ModuleName, DisablePopup);
...
if ParamStr(1) = '/U' then begin
  RestoreAllVectors;                    {needed to undo OpSwap's init code}
  Ifc := ModulePtrByName(ModuleName);   {get the IfcPtr for this module}
  if Ifc <> nil then begin              {make sure it is already installed}
    Ifc^.CmdEntryPtr;                   {call the CmdEntryPtr to unload}
    if LongInt(Ifc^.UserData) = 1 then {check status of Unload attempt}
      WriteLn(ModuleName, ' unloaded')
    else
      WriteLn('Unable to unload ', ModuleName);
  end else
    WriteLn(ModuleName, ' not installed, so it cannot be unloaded!');
end;
```

Code that allows a program to unload an already resident copy of itself when invoked with the command line option /U.

See Also

DisableTSR (OPTSR)
InstallModule
ModulePtrByName

InstallModule / MaxParagraphsToKeep

Declaration

```
procedure InstallModule(var ModName : String; CmdEntryPtr : PopupProc);
```

Purpose

Installs program as a resident module that can be located and accessed by other programs.

Description

This routine is similar to the routine by the same name in OPTSR. The only difference is the way the routine passed as the CmdEntryPtr is declared: for swappable TSRs it must be a FAR procedure without parameters.

OPSWAP1 predefines two identifiers that may be passed as the CmdEntryPtr parameter to InstallModule: NoExternalIfc and DisablePopUp. NoExternalIfc is passed when the module is to have no CmdEntryPtr associated with it. DisablePopup is used for unloading swappable TSRs.

Example

```
const MyTsrName : String[5] = 'MYTSR';
...
InstallModule(MyTsrName, DisablePopup);
```

See Also

DisablePopup
InstallModule (OPTSR)
ModuleInstalled

Declaration

```
function MaxParagraphsToKeep : Word;
```

Purpose

Maximum number of paragraphs to keep when going resident.

Description

This function is used internally to determine the maximum legal value of the ParasToKeep parameter passed to StayResSwap. If a program has been compiled with a max heap setting of 640K (the default), then MaxParagraphsToKeep will return the amount of memory that was available before the program was loaded. Hence, by subtracting ParagraphsToKeep (or the value to be passed to StayResSwap) from MaxParagraphsToKeep, you can calculate in advance approximately how much memory will be available after you go resident.

Example

```
StayResSwap(MaxParagraphsToKeep, 0, SwapName1, SwapName2, True);
WriteLn('Error going resident');
```

Go resident with largest possible heap.

See Also

ParagraphsToKeep (OPTSR)
StayResSwap

Declaration

```
procedure SetSwapMsgAttr(Attr : Byte);
```

Purpose

Sets the display attribute for swap messages.

Description

Attr is the new display attribute to use (default is $70, reverse video).

Example

```
if OpCrt.UseColor then SetSwapMsgAttr($1F);
```

Use White on Blue for color systems.

Declaration

```
procedure SetSwapMsgOn(On : Boolean);
```

Purpose

Turns swap message on or off.

Description

If the program is fairly small, then the swapping messages are undesirable. If the program is really large (like the DESKPOP sample program), then it is advisable to leave swapping messages turned on even if swapping goes to EMS. Swapping messages are on by default.

Example

```
if WillSwapUseEMS(TotalParasWeNeed) then begin
  WriteLn('Using EMS for swap');      {yes, so report to user}
  SetSwapMsgOn(False);                {disable swap messages}
end;
```

Declaration

```
procedure SetSwapMsgRow(Row : Byte);
```

Purpose

Sets the row where swapping messages are to appear.

Description

Row is the absolute screen row to display swapping messages. The default value is row 25.

For programs running on systems with displays capable of more than 25 rows, a Row value of $FF may be specified. This requests the swap message to appear on the bottom row of the screen even if the user is switching between 25 line mode and 43/50 line mode. This works for MCGA-, EGA-, and VGA-compatible video boards. OPCRT's CurrentDisplay variable contains the current display type.

Example

```
case CurrentDisplay of
  MCGA, EGA, VGA : SetSwapMsgRow($FF);
end;
```

If display is capable of greater than 25 rows, tell swap manager to use last row.

SetUserExitProc

Declaration

```
procedure SetUserExitProc(UserExitProc : PopupProc);
```

Purpose

Sets a pointer to a far procedure that is to be called before disabling the TSR.

Description

In some cases, a swappable TSR may have some housekeeping to do before unloading. Unfortunately, you can't use a Turbo Pascal ExitProc, because a TSR never actually halts. To provide some of the same functionality, DisableTSR (also called indirectly by DisablePopup) calls a single user exit procedure. You specify the procedure to be called by with SetUserExitProc. The specified UserExitProc must be compiled with the FAR model, it must be global, and it must have no parameters. It will be called just before the TSR is unloaded.

Example

```
{$F+}
procedure CleanHouse;
begin
  CloseAllOpenFiles; {flush and close any files we have open}
end;
{$F-}
...
{install the module}
InstallModule(ModuleName, DisablePopup);
{set up exit procedure for disable}
SetUserExitProc(CleanHouse);
```

See Also

DisablePopup DisableTSR

Declaration

```
procedure StayResSwap(ParasToKeep : Word; ExitCode : Byte;
                      SwapFileName1, SwapFileName2 : String;
                      UseSwapping : Boolean);
```

Purpose

Terminate and stay resident with optional TSR Swapping.

Description

ParasToKeep and ExitCode are interpreted just like the parameters of the same name in StayRes (OPTSR).

SwapFileName1 and SwapFileName2 specify the files used for swapping. (They are ignored if EMS is used.) They must provide complete DOS file specifications including drive, directory, and filename. They must not refer to the same file.

If UseSwapping is False, then the TSR does not use swapping and retains ParasToKeep paragraphs of memory. If UseSwapping is True, then only the swap manager code remains resident; the application will be swapped in when its hotkey is pressed.

If this procedure is successful in making the program resident, it never returns. If it does return, then there was an error going resident.

Example

```
{go resident}
StayResSwap(TotalParasWeNeed, 0, SwapName1, SwapName2, True);
{if we get here, then something went wrong}
WriteLn('Error going resident');
```

See Also

StayRes (OPTSR)

Declaration

```
function WillSwapUseEms(ParasToKeep : Word) : Boolean;
```

Purpose

Returns True if OPSWAP will use EMS memory for swapping.

Description

For accurate results, ParasToKeep must match the value passed for the parameter of the same name to StayResSwap.

Example

```
{calculate total heap now in use plus add 8K}
TotalParasWeNeed := ParagraphsToKeep + (8192 div 16); {extra 8K bytes of heap}
{now determine if EMS is to be used by Swap manager. If so, tell user}
if WillSwapUseEMS(TotalParasWeNeed) then begin
  WriteLn('Using EMS for swap');
  SetSwapMsgOn(False);  {turn swapping messages off when swapping to EMS}
end;
{go resident}
StayResSwap(TotalParasWeNeed, 0, SwapName1, SwapName2, True);
{if we get here, then something went wrong}
WriteLn('Error going resident');
```

F. OPSWAP: Swappable TSRs, Part 2

OPSWAP is a small unit that contains the swap management code. This unit must be listed last in the Uses list of the main program source file, and only there. This restriction guarantees that OPSWAP will be linked in the proper place within the resulting EXE file.

OPSWAP interfaces only a single identifier that is of use to applications programmers. See the "Overview of Swappable TSRs" for more background information.

Declaration

```
procedure SetPopTicker(TimeOut : Word);
```

Purpose

Sets the pop ticker for the CmdEntryPtr procedure.

Description

Most swappable TSRs will have no need to call this routine. It is generally useful only within assembly language interrupt service routines. The following assembly language code is adapted from the ALARM example on the BONUS disk.

Example

```
CODE    SEGMENT BYTE PUBLIC

        ASSUME  CS:CODE,DS:NOTHING
        EXTRN   SetPopTicker:Far
        ...
        MOV     AX,182          ; timeout parameter = 10 seconds
        PUSH    AX              ; pass timeout to SetPopTicker
        CALL    SetPopTicker    ; call SetPopTicker in OPSWAP
        ...
```

See Also

InstallModule

SetPopTicker (OPTSR)

G. OP8087: 8087 Support

As noted earlier, OP8087 is intended primarily for TSRs that use 8087 reals (Single, Double, Extended, Comp) for real number calculations. It does contain a function (Installed8087) to test for the presence of a math chip, but it is rarely needed because Turbo Pascal provides floating point emulation. Save8087, Restore8087, and Reinit8087, however, are of critical importance if your TSR needs to use the coprocessor. Although the popup dispatchers in OPTSR and OPSWAP take care of the CPU registers for you, they do not save the coprocessor's registers. So, if your popup uses 8087 reals and is compiled with {$N+,E-} (no emulation), the main routine should look something like this:

```
{$F+}
procedure PopupEntryPoint;
var Save8087Regs : SaveBuffer8087;
begin
  Save8087(Save8087Regs);
  Reinit8087;
  ...
  Restore8087(Save8087Regs);
end;
```

If your TSR or ISR uses the floating point emulator, you'll need to use two other routines: ReinitEmulator, which reinitializes Turbo Pascal 5's floating point emulator, and UseEmulator, which determines whether or not the floating point emulator is being used. The most important thing to keep in mind when using them is that ReinitEmulator should be called only after you have switched to an alternate stack. (In a popup, the stack switch would occur automatically; in an interrupt service routine, you would need to use SwapStackAndCall or SwapStackAndCallNear.) What ReinitEmulator does is to reinitialize the 230-byte data area starting at SSeg:0 that is used by the emulator, something that *must* be done before you can execute any arithmetic instructions using the emulator. Obviously, your alternate stack must be at least 230 bytes larger than would otherwise be necessary, in order to make room for the emulator's data.

What follows is a simple program that demonstrates the use of these routines. The program captures interrupt $1C (called at every clock) and performs floating point math each time an interrupt occurs.

```
{$N+,E+}
program UseEmulator;
uses  Op8087, OpInt;
const D : Double = 2.0;
var
  I : Word;
  UseEmu : Boolean;
  AltStack : array[1..2048] of Byte;

  procedure Dummy(var Regs : IntRegisters);
  var
    SB : SaveBuffer8087;
    D : Double;
  begin
    if UseEmu then
      {reinitialize emulator}
      ReinitEmulator
    else begin
      {save 8087 registers}
      Save8087(SB);
```

```
    {reinitialize 8087}
    Reinit8087;
  end;

  {do some math}
  D := 1.0;
  D := D * 2.0;

  if not UseEmu then
    {restore 8087 registers}
    Restore8087(SB);
end;

procedure Int1C(BP : Word); interrupt;
const
  Active : Boolean = False;
var
  Regs : IntRegisters absolute BP;
begin
  if not Active then begin
    Active := True;
    SwapStackAndCallNear(Ofs(Dummy), @AltStack[2048], Regs);
    Active := False;
  end;
end;

begin
  UseEmu := UseEmulator;
  if UseEmu then
    WriteLn('Using emulator...')
  else
    WriteLn('Using 80x87 chip...');
  {install $1C handler}
  if not InitVector($1C, 20, @Int1C) then
    Halt;
  WriteLn('Multiplying...');
  for I := 1 to 2000 do
    if D*D <> 4.0 then begin
      WriteLn('Error at ', I);
      Halt(1);
    end;
  WriteLn('Done...');
end.
```

A similar strategy is required for a TSR. Before going resident, you would call UseEmulator and store the result in a global variable. Within your popup routine(s), you would then check this variable: if the emulator is in use, you would call ReinitEmulator on entry to the popup; if not, you would call Save8087 and Reinit8087 on entry, and Restore8087 on exit.

By default, Turbo Pascal will produce a fatal runtime error whenever a numeric coprocessor calculation goes awry. Divide by zero, numeric overflow, square root of a negative argument—these and other errors will stop your program in its tracks. Even the OPERRHAN unit cannot help because by the time its error handler gains control, the 80x87 internal stack has been corrupted irreversibly.

Fortunately, OP8087 offers the means to solve this problem with just two simple routines. The Exceptions8087 procedure takes one boolean parameter which programs the coprocessor to produce exceptions (which lead to runtime errors) or to report errors via the coprocessor status word and numeric results. The second routine, Error8087, lets your program poll the

coprocessor status word to see whether the preceding stream of calculations has produced a numeric error. The Error8087 routine returns a result similar to that of the standard IoResult function: any non-zero value implies an error (which can be decoded for meaning), and the process of reading Error8087 resets the status register for further calculations.

It is important to note here that OP8087 behaves slightly differently depending on how it is compiled. If it is compiled with $N-, all the routines in the unit use true 8087 instructions. If it is compiled with $N+, all the routines use emulated 8087 instructions, which the SYSTEM unit will either translate into true 8087 instructions or pass on to the floating point emulator. In the remainder of the chapter, all references to the 8087 should therefore be assumed to refer also to Turbo Pascal's floating point emulator where applicable.

Types

```
SaveBuffer8087 =
  record
    Control  : Word;     {control word}
    Status   : Word;     {status word}
    Tag      : Word;     {tags for stack}
    InstPtr  : LongInt; {instruction pointer}
    OpPtr    : LongInt; {data pointer}
    Regs8087 : array[0..7] of Extended; {stack}
  end;
```

Used to store data needed to save and later restore the state of the math coprocessor using Save8087 and Restore8087.

Declaration

```
function Error8087 : Word;
```

Purpose

Return (and clear) the error status of the 80x87.

Description

This function can be used to check for errors after arithmetic operations performed with the math coprocessor. It is useful only if you have used Exceptions8087 to suppress exception interrupts. It returns the contents of the six lowest bits in the coprocessor's status word, those which indicate error conditions, and resets these flags to enable further processing.

If the function result is 0, no error occurred. Otherwise, you can determine what exception was generated by testing individual bits in the word that is returned:

Bit	Val	Meaning
5	32	Precision lost (result not exact--had to be rounded)
4	16	Underflow error
3	8	Overflow error
2	4	Divide by zero error
1	2	Denormalized operand (result so small significand has leading zeros)
0	1	Invalid operation (e.g., SQRT or LN of negative number)

Note: Bits 5 and 1 will often be set in cases you probably wouldn't consider errors. If desired, you can simply mask out these bits by ANDing the function result with $1D.

Example

```
Exceptions8087(False);
...
Extended3 := Extended1 * Extended2;
ErrorCheck := Error8087 (and $1D) ;
```

The call to Error8087 loads ErrorCheck with the contents of the coprocessor's status word and clears its error flags.

Declaration

```
procedure Exceptions8087(On : Boolean);
```

Purpose

Turn exception interrupts on or off.

Description

This procedure allows you to program a math coprocessor to produce exception interrupts, which generate runtime errors, or to report arithmetic errors using the coprocessor's status word. Since the 80x87 produces exception interrupts by default, the routine thus gives you a means of recovering from otherwise fatal runtime errors. Once you have disabled exception interrupts, you must check for errors manually by polling Error8087.

Note: The FINIT and FNINIT instructions used by routines such as Installed8087 (compiled with $N-) to test for the presence of an 80x87 reprogram the coprocessor to generate exception interrupts. If you execute another program after disabling exceptions, be sure to call Exceptions8087 after the Exec call, in case the program reprogrammed the coprocessor.

Examples

```
Exceptions8087(False);
...
Code := ExecDos('', True, NoExecDosProc);
Exceptions8087(False);
```

The second call ensures that the coprocessor is programmed properly after the Exec call.

Installed8087 / Reinit8087 / ReinitEmulator

Declaration
```
function Installed8087 : Boolean;
```

Purpose
Returns True if an 80x87 coprocessor is installed.

Description
This function allows you to determine whether a particular machine has a math coprocessor installed. Since the Turbo runtime library has its own test for the presence of an 80x87, and refuses to load programs that need it if none is installed, Installed8087 is rarely needed. In fact, if OP8087 is compiled with $N+, Installed8087 simply uses the SYSTEM unit's Test8087 variable to determine whether or not a math chip is installed.

Installed8087 is especially useful in programs used to configure programs that require an 80x87. In order to use it in such an application, OP8087 must be compiled with $N-.

Example
```
if not Installed8087 then begin
  WriteLn('The program being installed requires a math coprocessor!');
  Halt(1);
end;
```

Declaration
```
procedure Reinit8087;
```

Purpose
Reinitialize the 8087 chip.

Description
This routine reinitializes the 80x87 chip. It should be called just after a call to Save8087.

Example
See the introduction to this section for examples.

See Also
Save8087

Declaration
```
procedure ReinitEmulator;
```

Purpose
Reinitialize the emulator.

Description
This routine reinitializes the floating point emulator. It should be called only if UseEmulator returns True.

Example
See the introduction to this section for an example.

See Also
UseEmulator

Declaration
```
procedure Restore8087(var SaveBuf : SaveBuffer8087);
```

Purpose
Restores the 80x87 registers from the save buffer.

Description
This procedure is useful primarily in popup routines written with OPTSR or OPSWAP. It allows you to restore the state of the coprocessor (i.e., its registers and stack) so that arithmetic operations performed by the popup do not affect the program that was interrupted. Each call to Restore8087 must be preceded by a call to Save8087, which saves the state of the coprocessor.

Example
See the introduction to this section for examples.

Declaration
```
procedure Save8087(var SaveBuf : SaveBuffer8087);
```

Purpose
Saves the 80x87 registers in the save buffer.

Description
This procedure is useful primarily in popup routines written with OPTSR or OPSWAP. It allows you to save the state of the coprocessor so that the popup can use it without affecting the program that was interrupted. Each call to Save8087 must be followed later with a call to Restore8087, which restores the state of the coprocessor.

Example
See the introduction to this section for examples.

Declaration
```
function UseEmulator : Boolean;
```

Purpose
Return True if ReinitEmulator should be used rather than Reinit8087.

Description
This routine returns True if the floating point emulator is present *and* emulation is enabled. The most common use for UseEmulator is to determine if ReinitEmulator should be used rather than Reinit8087.

Example
See the introduction to this section for an example.

See Also
ReinitEmulator

H. Sample Programs

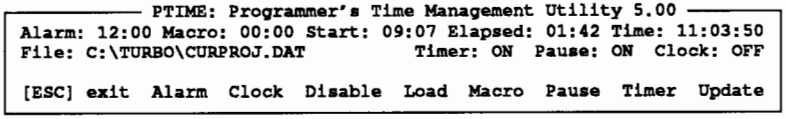
There are five sample TSRs included with **Object Professional**: SMACS and PTIME (which use OPTSR), and PSCREEN, POPHELP, and DESKPOP (which use OPSWAP). Of these, only PTIME is documented here, however. SMACS is documented at the end of Chapter 13 (Keyboard Macros), because of its extensive use of OPMACRO and OPMACED. PSCREEN is documented at the end of Chapter 4 (Window Objects), because it is a useful utility to have when working with PackedWindow objects. POPHELP is documented in Chapter 1 (Introduction), because it is of general interest to anyone who uses **Object Professional**. DESKPOP is also documented in Chapter 1, because it demonstrates so many of the features of **Object Professional**.

PTIME: Programmer's Time Management Utility

Most professional programmers need to keep track of the time that they spend on each project. But doing so can sometimes be difficult when you've got several projects going at once, especially if you get interrupted frequently. If this problem is yours--and perhaps even if it isn't--you may find PTIME a useful tool.

PTIME allows you to keep accurate records of how you spend your time. You can toggle timing on and off quickly and easily, use the pause feature to account for interruptions, save and load data files, set an alarm, activate an on-screen clock--you can even schedule an SMACS macro for playback while you're away from your desk. And because PTIME is memory resident, you don't need to stop what you're doing to use it.

Once PTIME is loaded into memory, you can activate it at almost any time by pressing <Ctrl><RightShift><T>. When you do, you'll see a display that looks something like this:


PTIME: Programmer's Time Management Utility 5.00
Alarm: 12:00 Macro: 00:00 Start: 09:07 Elapsed: 01:42 Time: 11:03:50
File: C:\TURBO\CURPROJ.DAT Timer: ON Pause: ON Clock: OFF
[ESC] exit Alarm Clock Disable Load Macro Pause Timer Update

The top line of the display shows you, from left to right, the time at which an alarm is scheduled to go off, the time at which a macro is scheduled to be played back, the time at which you started timing your work, the amount of time that has elapsed since then (less any time during which the Pause feature was activated), and the current time. The second line shows you the name of the current data file, and the status of the Timing, Pause, and on-screen Clock toggles. The bottom line shows you a menu of available commands, described below.

Alarm
When you press <A>, you will be asked to

```
Enter the time in "HH:MM" format (HH = 0-23, MM = 0-59):
```

If you wanted to schedule an alarm for 1 p.m., for example, you would enter

```
13:00
```

To cancel the alarm before it is set, simply press <Esc> when prompted for the time. If the alarm has already been set, it can be cancelled later by pressing <A> a second time and entering <Y> to confirm that it should be cancelled.

Clock
Pressing <C> activates the on-screen clock feature. When you leave PTIME, the current time will then be displayed roughly three times a second. To turn off the clock, pop up PTIME and press <C> a second time. Note that the on-screen clock is always disabled temporarily while PTIME is popped up.

Disable
This feature allows you in many cases to unload PTIME from memory without rebooting the system. Because doing so is potentially dangerous, PTIME will not disable itself if any program loaded above it uses the same interrupt vectors.

Load
When you press <L>, you will be asked to enter the name of a file in which to store timing data. If it does not exist, a new data file will be created. Otherwise, PTIME will simply store the name of the file for future reference. This command is not available while the timer is on.

Macro
This feature allows you to schedule an SMACS macro for playback. If SMACS is not loaded when you press <M>, an error message will be displayed. Otherwise you will be asked first to press the key corresponding to the macro, then to enter the time at which the macro is to be played back. For example, if you wanted to play back a macro associated with <AltP> at 5:30 p.m. you would press <AltP> and enter

```
17:30
```

As with the Alarm command, you can cancel the playback of the macro before it has been scheduled by pressing <Esc> when prompted for the time, and you can cancel it later by pressing <M> a second time.

Pause
The Pause command allows you to account for brief interruptions in your work. If the timer is on and you get interrupted, you can temporarily deactivate the timer by pressing <P>. Although the timer will remain on, the elapsed time will remain unchanged until the Pause toggle is pressed a second time.

Timer
This is the most fundamental of the PTIME commands. When you press <T>, the program turns on its timer to begin tracking elapsed time and appends a new record to the current data file. If no data file has yet been selected with the Load command you will be prompted for one. To turn off the timer, press <T> a second time. The last record in the data file will then be updated.

Update
The Update command exists mainly to allow you to recover from any system crashes that might occur while the timer is on. If this happens, you can simply reload PTIME after rebooting, pop up the program, press <U> for Update, and enter the name of the data file. After PTIME has updated the last record in the file, you can then resume timing by issuing the Timer command. The only data that could be lost due to a system crash is that concerning the duration of Pauses in your work. Since that is the case, you may wish to use Update at the end of each Pause. Note that you can also recover from crashes manually by editing the last record in the data file with a text editor (see the discussion of this below).

Format of a PTIME Data File

PTIME stores its data in a file containing ASCII text stored in fixed-length records. Each record has two lines associated with it: one for the timing data, and another for the comment. The comment line may be blank, but it will still be padded to its maximum length with spaces. Each line is terminated by a carriage return and line feed. The first record in the file is a header indicating that the file is a PTIME data file.

The exact format of the remaining records in the data file is as follows:

```
mm/dd/yy hh:mm - mm/dd/yy hh:mm  elapsed: hh:mm  update allowed: x
comment line
```

The first date/time pair indicate when the timer was started, the second when it was stopped. The elapsed time is the number of hours and minutes that elapsed between the two. The "update allowed" field will contain either 'Y' or 'N'. 'Y' indicates that an update is allowed because the timer has not yet been stopped. 'N' indicates that the record is complete and need not be updated. This flag is used internally by the Update option to determine whether a record needs updating. The second line has the comment entered by the user when the timer was stopped. The comment may be up to 76 characters line. Both lines are padded to 76 characters, and a carriage return and line feed are added to the end of each line.

Those of you familiar with the Turbo Professional version of PTIME will note that timing data is stored very differently in the **Object Professional** version. The purpose for the change was to eliminate the need for the PTRPT program. Since the data is now stored as ASCII text rather than binary data, you do not need a separate program to view the data contained within it.

Although you can also edit the data in the file, you must *be careful not to modify the record length of any existing record* if you intend to use the file to record more data (each line is 78 characters including the CR/LF). Many text editors truncate trailing spaces for lines of text. You should not use such an editor on PTIME data files. Probably the easiest solution is to "retire" a data file after it has been edited and start a new one. If you do change the record length of one of PTIME's data files, and then try to use it again, PTIME will probably overwrite existing data, since it will calculate the wrong place to seek to in the file when it is time to append a new record.

12. Memory Allocation

A. Overview

Much has been written about "the 640K barrier" imposed by MS-DOS, and most of us have encountered it ourselves. Sophisticated software needs plenty of memory for code and especially data, but what can you do when you run out of room? Fortunately, there are two alternatives to using normal RAM.

One of these, extended memory, is available only on '286, '386, and '486 machines. Extended memory exists above the one megabyte mark and is accessible only by switching these machines into protected mode.

The other alternative, expanded memory, is available on any IBM-compatible machine equipped with a board designed in accordance with the Expanded Memory Specification, or EMS for short. EMS memory is made accessible by the Expanded Memory Manager (EMM) software driver, and several companies have developed drivers that allow extended memory to emulate expanded memory. Unfortunately, Turbo Pascal, like most compilers, does not provide you any way to use either extended or expanded memory to store your data. **Object Professional** does.

The OPEMS unit contains a variety of routines for working with expanded memory. It allows you to determine whether or not EMS memory is installed and, if so, how much is currently available. You can allocate and deallocate memory, and you can also save and restore the state ("context") of the expanded memory manager, in case you need to access EMS memory from inside a TSR, for example.

We should probably point out here that OPEMS is designed to work with version 3.x or higher of the Expanded Memory Specification. It does not currently support any of the newer functions added in EMS 4.0.

EMS memory is allocated in 16K blocks known as pages. This makes it quite useful for storing medium-sized arrays, linked lists, and binary trees, but it is less than ideal for general purpose storage of smaller pieces of data that need to be accessed quickly on a regular basis. If your dynamic storage needs are minimal, you would be better off allocating memory in normal RAM using New or GetMem. Although EMS memory is also less than ideal for storing large arrays, **Object Professional** offers a solution to this problem in the related unit, OPLARRAY. Using OPLARRAY, you can easily work with arrays of up to 8 megabytes. See the documentation for OPLARRAY in Chapter 8, Section C for more details.

Similarly, the OPEXTMEM unit gives you the tools you need to use extended memory for your data storage needs. These routines allow you to access extended memory under DOS 3.x or 4.x without colliding with VDISK.SYS, MicroSoft's RAMDRIVE.SYS, or with any other program compatible with either of the two. To access the memory, you need to take the following steps:

1. Find out how much extended memory is available by calling ExtMemAvail.
2. If the amount of memory available is sufficient, use GetExtMem to allocate the total amount that you will use.
3. Once the memory has been allocated, use MoveExtMem to move data from normal RAM to extended memory and back again.

As you may have guessed already, OPEXTMEM contains extended memory counterparts for many of Turbo Pascal's built-in routines: MemAvail, GetMem, FreeMem, Mark, Release, and

Move. Although the names are similar, there are important differences between them, and a few simple guidelines must be kept in mind when using these routines.

First and most important is the fact that extended memory must be allocated and deallocated in sequential order. If you allocate three blocks of memory, and release the middle block by calling FreeExtMem, all memory above the first block will appear to be available. (The reasons for this restriction will be immediately obvious if you study the source code for OPEXTMEM.)

The second most important guideline to remember is that all addresses in extended memory are referenced using linear, rather than segmented, notation. For example, the segmented address $0040:$0012 corresponds to the linear address $00000412. The slight disadvantage of this method is that, when moving data between normal and extended memory, you must translate segmented addresses into linear addresses, or huge pointers, by calling the PtrToHuge function. The great advantage of it is that huge pointers are nothing more than long integers, so you can create arrays much larger than 64K in extended memory and use long integers for indexing into them. For example, if HP is a huge pointer to a 0-based array, then you can calculate the address of the N'th element (HPn) thus:

```
HPn := HugePtr( LongInt(HP) + (LongInt(N) * SizeOf(Element)) );
```

Third, GetExtMem allows you to allocate memory only in 1K blocks (1,024 bytes). As with EMS memory, this makes extended memory less convenient to use for general purpose storage, but it also means that you can allocate several megabytes of contiguous memory with a single call to GetExtMem. In addition, the memory manager in older versions of RAMDRIVE.SYS can handle only 95 blocks of extended memory, at least 1 of which will always be used as a RAM disk. Although OPEXTMEM allows you to allocate more than 94 blocks (so long as RAMDRIVE.SYS is not present), you should not count on having access to more than that if you want your programs to be compatible with RAMDRIVE.

Fourth, the MoveExtMem procedure that is used to transfer data between normal and extended memory does not allow you to move data in bytes, only in words (two bytes). This means that you must be extra careful in choosing your data structures, and that you will sometimes need to pad record types, for example, with dummy bytes to insure that their sizes are even numbers. The one benefit of this restriction is that you can move up to 128K of contiguous data with a single call to MoveExtMem.

Fifth, in order to accommodate RAMDRIVE.SYS, most of the routines in OPEXTMEM require a little over 1K of stack space for local variables. You should keep this in mind when using the unit in programs with restricted stack space.

A final important consideration to keep in mind is that OPEXTMEM works only with DOS versions 3.x and 4.x. This restriction is intended to avoid potential conflicts when running a program in the OS/2 "compatibility box." The fact that OPEXTMEM cannot be used to allocate memory when running under DOS 2.x is irrelevant, since 80286, 80386, and 80486 machines (the only kind that might have extended memory installed) cannot run DOS 2.x.

Since OPEXTMEM was written, Microsoft has released a new specification and driver for extended memory called XMS (eXtended Memory Specification). At this time, OPEXTMEM does not support XMS.

B. OPEMS: EMS Memory Management

This section describes all of the types, variables, constants, procedures and functions found in the interface section of the OPEMS unit.

Constants

```
EmsErrorCode = $0FFFF;
```

Used by several functions in OPEMS to signal an error condition.

AllocateEmsPages

Declaration

```
function AllocateEmsPages(NumPages : Word) : Word;
```

Purpose

Allocates the indicated number of pages and returns a handle.

Description

This function allows you to allocate expanded memory in 16K (16,384 bytes) blocks, known as pages. You might think of it as being like a combination of GetMem, used to allocate normal memory in Turbo Pascal, and DOS's open file function, which returns a file handle. As with DOS file handles, the handle returned by AllocateEmsPages is your ticket to accessing the memory you've allocated. In particular, it is needed to map a particular page into normal memory (using MapEmsPage) and to deallocate the memory when it is no longer needed (using DeallocateEmsHandle).

AllocateEmsPages returns EmsErrorCode in case of error. The most common cause of error is insufficient memory to fill the request. You can determine the number of pages available at any given time by calling EmsPagesAvail.

Note on handles: Although the standard version of the Expanded Memory Manager assigns handles in the order $FF00, $FE01, $FD02, etc., this order is not guaranteed to be correct for all versions of the EMM driver. At least one EMM driver we are aware of returns handles in the order 0, 1, 2, etc.

Example

```
PagesToAllocate := 4;
if (EmsPagesAvail >= PagesToAllocate) then begin
  MyEmsHandle := AllocateEmsPages(PagesToAllocate);
  if (MyEmsHandle = EmsErrorCode) then
    {error};
end else
  {insufficient memory};
```

This example shows how to allocate 64K (4 16K pages) of EMS memory using AllocateEmsPages. Notice that EmsPagesAvail is called first to insure that sufficient memory is available.

See Also

DeallocateEmsHandle
EmsPagesAvail
MapEmsPage

Declaration

```
function DeallocateEmsHandle(Handle : Word) : Boolean;
```

Purpose

Deallocates the indicated handle and the memory associated with it.

Description

This function is the EMS equivalent of Dispose and FreeMem. It allows you to deallocate memory previously allocated with AllocateEmsPages. Note that only the handle returned by AllocateEmsPages is needed, and not the number of pages assigned to it. DeallocateEmsHandle returns False if the handle cannot be deallocated. The most likely cause of error is an invalid handle, or a handle that has already been deallocated.

Very important: Unlike OPEXTMEM, OPEMS does not automatically deallocate the EMS memory you've allocated, primarily because that convenience is not worth the overhead required to provide it. So, any program that uses OPEMS should contain an exit handler that will deallocate any EMS memory you've allocated. A simple example is shown below.

Examples

```
MyEmsHandle := AllocateEmsPages(PagesToAllocate);
...
if not DeallocateEmsHandle(MyEmsHandle) then {error};
```

Demonstrates how to deallocate memory previously allocated with AllocateEmsPages.

```
{$F+}
procedure MyExitProc;
begin
  ExitProc := SaveExitProc;
  if not DeallocateEmsHandle(MyEmsHandle) then {error};
end;
{$F-}
...
SaveExitProc := ExitProc;
ExitProc := @MyExitProc;
```

A simple exit handler that deallocates all memory assigned to MyEmsHandle.

See Also

AllocateEmsPages

Declaration

```
function EmmSigFound : Boolean;
```

Purpose

Returns True if EMM signature found.

Description

Like EmsInstalled, this function tries to determine whether or not the Expanded Memory Manager (EMM) is installed. The method it uses is quite different, however. It checks first to see if the interrupt vector used by the EMM ($67) is nil. If so, it returns False. If not, it looks at offset 10 of the segment pointed to by the INT $67 vector for the EMM's "signature," an 8-byte array containing "EMMXXXX0". If the signature is found, EmmSigFound returns True.

Although this method for detecting the EMM is widely used (Turbo Pascal's overlay manager uses it, for example), we recommend that you use EmsInstalled instead if at all possible. Several of the early EMS drivers fail the "signature test" used by EmmSigFound, but pass the "handle test" used by EmsInstalled.

See Also

EmsInstalled

EmsActiveHandles / EmsInstalled

Declaration

```
function EmsActiveHandles : Word;
```

Purpose

Returns the number of active EMS handles.

Description

This function allows you to determine the number of active EMS handles that have been assigned by the Expanded Memory Manager. Like EmsTotalPages, it is useful primarily for displaying status information. EmsActiveHandles returns EmsErrorCode ($FFFF) in case of error. The most likely causes of error are the same as those for EmsStatusOK.

Example

```
WriteLn(LongInt(EmsTotalPages)*16384, ' bytes of EMS memory installed.');
WriteLn(LongInt(EmsPagesAvail)*16384, ' bytes available.');
WriteLn(EmsActiveHandles,             ' EMS handles active.');
```

Here the call to EmsActiveHandles allows the program to display the number of handles that are active, as well as the more mundane status information.

See Also

EmsStatusOK EmsTotalPages

Declaration

```
function EmsInstalled : Boolean;
```

Purpose

Returns True if the EMM is installed.

Description

This function tells you whether or not an Expanded Memory Manager (EMM) is installed. It does not tell you if a memory board adhering to the Expanded Memory Specification (EMS) is installed. You should call this routine before using any of the other procedures and functions in the OPEMS unit. In rare cases, you might also need to call EmsVersion to insure that a particular EMS function that your program uses is available.

EmsInstalled uses the "handle test" for determining if the EMM is installed. That is, it tries to open "EMMXXXX0"; if that operation succeeds, and "EMMXXXX0" is flagged as a character device, the function returns True. This is the most reliable test to determine whether the EMM is installed, but it may cause problems in rare situations. If you encounter problems in using it, call EmmSigFound instead.

Example

```
if not EmsInstalled then begin
  WriteLn('This program requires EMS memory. Aborting...');
  Halt(1);
end;
```

In this example, a program that requires EMS memory in order to run aborts if EmsInstalled returns False.

See Also

EmmSigFound EmsVersion

Declaration

```
function EmsPageFramePtr : Pointer;
```

Purpose

Returns the page frame base address as a pointer to the page frame.

Description

This function returns a pointer to the base address of the page frame used by the EMM. A page frame occupies an entire 64K segment, divided into four 16K pages, numbered 0-3. These four pages are called physical pages, as opposed to logical pages, because they exist within the 808x's one megabyte address space. Logical pages, by contrast, exist on the expanded memory board, and their numbers are relative to a given handle. If you allocate three pages to a particular handle with AllocateEmsPages, for example, you would refer to them individually as page 0, 1, or 2. In order to access the contents of a particular logical page, moreover, you must map the page into one of the four physical pages using MapEmsPage.

Perhaps the easiest way to understand the concept of a page frame is to think in terms of some simple data structures:

```
type EmsPage = array[1..16384] of Byte;
     EmsPageFrame = record
                      Page0, Page1, Page2 : EmsPage;
                      Page3 : array[1..16369] of Byte;
                    end;
     PageFramePtr = ^EmsPageFrame;
```

Notice that, because Turbo Pascal does not allow data types larger than 65521 bytes, Page3 is declared to be smaller than the first three pages. As long as range checking is off, however, we can safely refer to all 16384 elements in the array.

Now, to move 16K of data into the third physical page (#2), for example, you'd say something like

```
MyPageFramePtr := EmsPageFramePtr; {only need to do this once!}
Move(Source, MyPageFramePtr^.Page2[1], 16384);
```

Note that OPEMS does not define these types for you because there's no guarantee that you'll want to treat any given page as an array of byte.

Example

```
var MyPageFramePtr : PageFramePtr;
    MyEmsPagePtr : ^EmsPage;
...
MyPageFramePtr := EmsPageFramePtr;
if not MapEmsPage(MyEmsHandle, 0, 2) then {error};
MyEmsPagePtr := @MyPageFramePtr^.Page2[1];
```

This example shows how you could map the first logical page (#0) associated with MyEmsHandle into the third physical page (#2), and load MyEmsPagePtr with the starting address of the physical page.

See Also

AllocateEmsPages

MapEmsPage

EmsPagesAvail / EmsPagesOwned

Declaration

```
function EmsPagesAvail : Word;
```

Purpose

Returns number of available pages reported by the Expanded Memory Manager.

Description

This function allows you to determine how much expanded memory is available for your use. EmsPagesAvail returns the number of 16K (16,384 bytes) pages that are available, or EmsErrorCode in case of error. The most likely causes of error are the same as those for EmsStatusOK. To determine how many pages are actually installed in the system, you can use EmsTotalPages.

Example

```
PagesToAllocate := EmsPagesAvail;
if (PagesToAllocate = EmsErrorCode) then
  {error condition}
else if (PagesToAllocate <> 0) then
  MyEmsHandle := AllocateEmsPages(PagesToAllocate);
```

This example shows how to allocate all available EMS memory to your program.

See Also

EmsStatusOK EmsTotalPages

Declaration

```
function EmsPagesOwned(Handle : Word) : Word;
```

Purpose

Returns the number of pages owned by Handle.

Description

This function allows you to determine the number of pages allocated to a particular handle. It will return a value in the range 1-512 if successful, or EmsErrorCode ($FFFF) in case of error. The most likely cause of an error is an invalid handle. EmsPagesOwned is useful primarily for displaying status information.

Example

```
var
  Handle, Pages : Word;
  HandleRec : record
                LowByte, HighByte : Byte;
              end absolute Handle;
begin
  Handle := $FF00;
  repeat
    Pages := EmsPagesOwned(Handle);
    if (Pages <> EmsErrorCode) then
      WriteLn('Handle $', HexW(Handle), ': ', Pages, ' pages');
    Inc(HandleRec.LowByte);
    Dec(HandleRec.HighByte);
  until (Pages = EmsErrorCode);
end;
```

This example demonstrates how to determine how many pages have been allocated to each active EMS handle. (Not guaranteed to work, see the special note on handles in the documentation for AllocateEmsPages.)

See Also

AllocateEmsPages

Declaration

```
function EmsStatusOK : Boolean;
```

Purpose

Returns True if the EMM reports its status as being OK.

Description

EmsStatusOK allows you to test the expanded memory hardware to insure that it is functional. It returns True if the Expanded Memory Manager reports that its status is OK, False if it does not. The most likely causes of an error are corruption of EMM driver's code or a malfunction in the memory board. Note that EmsStatusOK should be used only after EmsInstalled has been called to establish that the EMM is installed.

Example

```
if not EmsInstalled then begin
  WriteLn('This program requires EMS memory. Aborting...');
  Halt(1);
end;
if not EmsStatusOK then begin
  WriteLn('Malfunction in EMS software or hardware. Aborting...');
  Halt(1);
end;
```

Here EmsInstalled is used to verify that the Expanded Memory Manager is installed, and EmsStatusOK is used to make sure that it is functioning properly.

See Also

EmsInstalled

Declaration

```
function EmsTotalPages : Word;
```

Purpose

Returns total number of pages of EMS memory, including allocated pages.

Description

This function returns the number of 16K pages of memory installed, or EmsErrorCode in case of error. The most likely causes of error are the same as those for EmsStatusOK. It is useful primarily for displaying status information.

Example

```
WriteLn(LongInt(EmsTotalPages)*16384, ' bytes of EMS memory installed.');
```

See Also

EmsPagesAvail

EmsVersion / MapEmsPage

Declaration

```
function EmsVersion : Byte;
```

Purpose

Returns a BCD version number of the EMM.

Description

This function returns a value in BCD (Binary Coded Decimal) format indicating the version of the Expanded Memory Manager. Version 3.2, for example, would be represented as $32. EmsVersion returns 0 in case of an error, usually indicating that the EMM is not installed. Note that all routines in OPEMS are available in versions 3.0 and higher of the EMM, so EmsVersion should rarely be needed.

Example

```
if EmsVersion < $32 then begin
  WriteLn('Expanded Memory Manager 3.2 or higher required.');
  Halt(1);
end;
```

Declaration

```
function MapEmsPage(Handle, LogicalPage : Word; PhysicalPage : Byte) : Boolean;
```

Purpose

Maps the specified LogicalPage associated with Handle into PhysicalPage (0-3).

Description

This function is used to make data stored in EMS memory accessible. (*Note:* If you haven't done so already, please read the entry for EmsPageFramePtr, which defines some of the terms used here.) What it does is to map a particular logical page (as defined by the Handle and LogicalPage parameters) into one of the four physical pages in the Expanded Memory Manager's page frame. This is what allows you to treat expanded memory as normal memory.

The Handle parameter is the value returned previously by AllocateEmsPages when you allocated your memory. LogicalPage is a number between 0 and n-1, where n is the number of pages allocated to Handle by AllocateEmsPages. PhysicalPage is a number between 0 and 3, representing one of the four physical pages that make up the page frame.

MapEmsPage returns True if successful, and False if the operation failed. The most likely cause of failure is an invalid handle or page number.

Example

```
type EmsPage = array[1..16384] of Byte;
     EmsPageFrame = record
                      Page0, Page1, Page2 : EmsPage;
                      Page3 : array[1..16369] of Byte;
                    end;
     PageFramePtr = ^EmsPageFrame;
var  MyPageFramePtr : PageFramePtr;
     MyEmsPagePtr : ^EmsPage;
...
  MyPageFramePtr := EmsPageFramePtr;
  if not MapEmsPage(MyEmsHandle, 0, 2) then {error};
  MyEmsPagePtr := @MyPageFramePtr^.Page2[1];
```

Demonstrates how to map the first logical page (#0) associated with MyEmsHandle into the third physical page (#2), and load MyEmsPagePtr with the starting address of the physical page. See the entry for EmsPageFramePtr for details.

See Also

AllocateEmsPages EmsPageFramePtr

Declaration

```
function RestoreEmsContext(Handle : Word) : Boolean;
```

Purpose

Restores the mapping context of the EMM driver for the handle specified.

Description

This function is intended primarily for use in popups that use EMS memory. In conjunction with SaveEmsContext, it allows you to save the mapping context of the Expanded Memory Manager on entry to the popup, and restore it when you exit, so that the interrupted program is not affected. The handle passed as a parameter should be the same one used in a prior call to SaveEmsContext. RestoreEmsContext returns True if restoration is successful.

Example

```
{$F+}
procedure MyPopup;
begin
  if not SaveEmsContext(MyEmsHandle) then Exit;
  {-- body of the popup --}
  if not RestoreEmsContext(MyEmsHandle) then
    {error--display warning};
end;
```

This example illustrates how to use RestoreEmsContext in a popup. The popup should save the context on entry, and restore it on exit.

Declaration

```
function SaveEmsContext(Handle : Word) : Boolean;
```

Purpose

Saves the EMM context for resident programs.

Description

This function is intended primarily for use in popups that use EMS memory. In conjunction with RestoreEmsContext, it allows you to save the mapping context (i.e., the page mapping registers on the memory board) of the Expanded Memory Manager on entry to the popup, and restore it when you exit, so that the interrupted program is not affected. The handle passed as a parameter must have been allocated previously using a call to AllocateEmsPages. SaveEmsContext returns True if successful.

Example

```
MyEmsHandle := AllocateEmsPages(PagesToAllocate);
{check for EmsErrorCode}
...
{$F+}
procedure MyPopup;
begin
  if not SaveEmsContext(MyEmsHandle) then Exit;
  {-- body of the popup --}
  if not RestoreEmsContext(MyEmsHandle) then
    {error--display warning};
end;
```

This example illustrates how to use SaveEmsContext in a popup. The popup should save the context on entry, and restore it on exit. MyEmsHandle is the handle previously assigned to the program with the initial call to AllocateEmsPages.

See Also

AllocateEmsPages

C. OPEXTMEM: Extended Memory Management

This section describes all of the types, variables, constants, procedures and functions found in the interface section of the OPEXTMEM unit.

Types

```
HugePtr = Pointer;
```

A HugePtr holds a 24-bit linear address. It can be cast to a LongInt (and vice versa) for indexing purposes, and you can test it for a value of nil, just like a normal pointer (which holds a segmented address).

Declaration

```
function ExtMemAvail : Word;
```

Purpose

Returns the amount extended memory, in KB, that is actually free.

Description

Much like Turbo's MemAvail, ExtMemAvail allows you to determine how much extended memory is available for your program's use. The number it returns, however, is in kilobytes (1,024 bytes), since that is the unit used by GetExtMem when allocating extended memory. The function returns 0 if DOS 3.x or 4.x is not being used, even if there is actually some extended memory available.

Example

```
if ExtMemAvail >= KBtoAllocate then
  MyHugePtr := GetExtMem(KBtoAllocate)
else begin
  WriteLn('Insufficient extended memory');
  MyHugePtr := nil;
end;
```

In this example, ExtMemAvail is called to see if sufficient memory is available before making an allocation request to GetExtMem.

See Also

GetExtMem

Declaration

```
function ExtMemTotal : LongInt;
```

Purpose

Returns total number of bytes of extended memory in the system.

Description

In contrast to ExtMemAvail, which returns the number of kilobytes of extended memory available, ExtMemTotal returns the total number of bytes of extended memory installed in the system. It is useful primarily for displaying status information. The function returns 0 if DOS 3.x or 4.x is not being used, even if there is actually some extended memory installed.

Example

```
WriteLn(LongInt(ExtMemAvail)*1024,
        ' bytes of extended memory available out of ', ExtMemTotal);
```

FreeExtMem / GetExtMem

Declaration

```
function FreeExtMem(P : HugePtr) : Boolean;
```

Purpose

Deallocate a block of extended memory.

Description

FreeExtMem deallocates a block of extended memory allocated with GetExtMem. It works only if the block to deallocate is the last block allocated. The function returns False if it was unable to deallocate the memory. Note that it is not necessary to call FreeExtMem at the end of your program, since OPEXTMEM will automatically deallocate any memory you have allocated.

Example

```
MyHugePtr := GetExtMem(KBtoAllocate);
...
if not FreeExtMem(MyHugePtr) then begin
  WriteLn('Unable to dispose of allocated memory');
  Exit;
end;
```

GetExtMem is called to allocate extended memory, FreeExtMem to dispose of the memory that was allocated.

Declaration

```
function GetExtMem(N : Word) : HugePtr;
```

Purpose

Allocate N KB of extended memory.

Description

Much like GetMem, GetExtMem lets you allocate a block of extended memory, but it can allocate memory only in 1K (1,024 byte) blocks. Since the function returns nil if there is insufficient memory to fill the request, it is not imperative that you call ExtMemAvail before making the request. A block of extended memory allocated with GetExtMem can later be disposed of using FreeExtMem.

Example

```
MyHugePtr := GetExtMem(KBtoAllocate);
if MyHugePtr = nil then begin
  WriteLn('Unable to allocate extended memory');
  Exit;
end;
```

See Also

ExtMemAvail

Declaration

```
procedure MarkExtMem(var P : HugePtr);
```

Purpose

Store the location of the next extended memory block in P.

Description

Like Turbo's Mark, MarkExtMem allows you to store the location of the next free memory block in a variable. You can then allocate multiple blocks of extended memory using GetExtMem, and dispose of all of them at once with ReleaseExtMem. Unlike Mark/Release and GetMem/FreeMem, however, MarkExtMem/ReleaseExtMem do not conflict with GetExtMem/FreeExtMem. P will be set to nil if there is no extended memory available.

Example

```
MarkExtMem(ExtMemStart);
HP1 := GetExtMem(KBtoAllocate1);
HP2 := GetExtMem(KBtoAllocate2);
...
MemoryReleased := ReleaseExtMem(ExtMemStart);
```

MarkExtMem is called before allocating several blocks of extended memory with GetExtMem, and ReleaseExtMem is called to dispose of it when it is no longer needed.

See Also

GetExtMem ReleaseExtMem

Declaration

```
procedure MoveExtMem(Src, Dest : HugePtr; WCount : Word);
```

Purpose

Move WCount words of extended memory from Src to Dest.

Description

Much like Turbo's Move, MoveExtMem allows you to move blocks of memory between any two locations in memory. That is, you can use it to transfer data from extended memory to normal memory (or vice versa), or between two regions of extended memory. Unlike Move, however, MoveExtMem does not allow you to move data in one-byte units--a very important caveat to keep in mind when using the routines in OPEXTMEM.

Note: When using Move with normal pointers, you must dereference the pointers:

```
Move(PointA^, PointB^, BytesToMove);
```

When using MoveExtMem with HugePtr's, however, you must *not* dereference them (using the '^' symbol). Also when transferring data from normal to extended memory and back again, the address in normal memory must first be converted from a segmented address (pointer) to a linear address (huge pointer, or HugePtr) using PtrToHuge. For example,

```
MoveExtMem(PtrToHuge(@MyArray[1]), MyHugePtr, 512);
```

would move 512 words (1,024 bytes) from MyArray[1] to the address pointed to by MyHugePtr.

Examples

```
MoveExtMem(PtrToHuge(MyBufferPtr), MyHugePtr, WordsToMove);
```

Correct. This would move WordsToMove words of data from the address in normal memory pointed to by MyBufferPtr to the address pointed to by MyHugePtr.

```
MoveExtMem(PtrToHuge(MyBufferPtr)^, MyHugePtr^, WordsToMove);
```

Incorrect! The huge pointers must not be dereferenced.

See Also

PtrToHuge

PtrToHuge / ReleaseExtMem

Declaration

```
function PtrToHuge(P : Pointer) : HugePtr;
```

Purpose

Converts a pointer to a huge pointer.

Description

PtrToHuge converts a regular pointer representing a segmented address to a huge pointer representing a linear address. Among other things, PtrToHuge is needed to move data between normal and extended memory using MoveExtMem. Note that the pointer passed to PtrToHuge must be in the range $0:$0 to $FFFF:$000F.

Example

```
MoveExtMem(PtrToHuge(MyBufferPtr), MyHugePtr, WordsToMove);
```

Declaration

```
procedure ReleaseExtMem(P : HugePtr);
```

Purpose

Release all extended memory above P.

Description

Much like Release, ReleaseExtMem lets you dispose of all memory blocks above a given point in memory, a point whose location was stored in a HugePtr variable using MarkExtMem. You can thus allocate multiple blocks of extended memory with GetExtMem, and dispose of all of them at once with ReleaseExtMem. Unlike Turbo's Mark/Release and GetMem/FreeMem, OPEXTMEM's MarkExtMem/ReleaseExtMem pair do not conflict with GetExtMem/FreeExtMem. Note also that it is not necessary to call ReleaseExtMem at the end of your program, since OPEXTMEM will automatically dispose of any memory you have allocated.

Very important: The P parameter must have been set by MarkExtMem. It cannot be a HugePtr returned by GetExtMem.

Example

```
MarkExtMem(ExtMemStart);
HP1 := GetExtMem(KBtoAllocate1);
HP2 := GetExtMem(KBtoAllocate2);
...
ReleaseExtMem(ExtMemStart);
```

MarkExtMem is called before allocating several blocks of extended memory with GetExtMem, and ReleaseExtMem releases them when they are no longer needed. Calling ReleaseExtMem would be unnecessary if the program were about to end.

See Also

MarkExtMem

13. Macros

A. Overview

The term macro is widely used in a number of ways. Many general purpose programs, notably spreadsheets, incorporate a macro language that allows you to tailor them for particular tasks. Similarly, some compilers and assemblers allow you to assign a symbolic name to a sequence of instructions; Turbo Pascal's "inline procedures" can be considered macros in this sense of the word. More frequently, however, the term refers to keyboard macros, a facility that allows you to make a single keystroke do the work of many. Most commercial text editors and word processors include such a facility, and now **Object Professional** allows you to incorporate this powerful feature into your own applications.

The OPMACRO unit contains all the low-level routines you should need to do this. They allow you to define macros, assign names to them, record them, play them back, save them to a file for future use, and retrieve them when they're needed again.

Once defined, keyboard macros can be played back in one of two ways. If macro processing is enabled (by a call to MacrosOn), a macro will automatically be played back any time that the corresponding key is pressed. Alternatively, you can force a particular macro to be played back by calling either StartMacro or StartMacroByKey. In either case, the task of playing back the macro is performed by OPMACRO's hidden macro engine, which can also record macros for you.

One important restriction is worth noting here. OPMACRO does not allow "nested macros." For example, if one macro (call it macro A) has been assigned to <F1>, a second one (call it macro B) to <F2>, and macro A contains <F2>, macro B will not be played back as part of macro A. There are two main reasons for this restriction. First, it avoids the possibility of an infinite loop, such as might occur if macro A in our example contained <F2> and macro B contained <F1>. Second, it allows OPMACRO to be used to remap the keyboard (e.g., you could use it to reverse the functions of <F1> and <F3>). (You can, however, "chain" macros together using the hook provided by SetEndOfMacroProc.)

Object Professional's keyboard macros are stored in variably sized records of type MacroRec. The first field in the record, NumKeys, is a word indicating the number of keys in the macro. The second field, KeyArray, is an array of words with a character code in the low byte and a scan code in the high byte, the same format used by the BIOS. The end of the array is marked by a word with the value EndOfMacro ($FFFF, or -1). You might think of this format as being like a cross between Turbo-style strings, which start with a length byte, and ASCIIZ strings, which use null characters (#0) to mark the ends of strings. Each macro is associated with a key, such as <F1>, and an optional name, such as 'Help'. By default, you are limited to 100 (MaxMacros) such macros of up to 1000 keystrokes (MaxKeysInMacro) each, but because **Object Professional** comes with complete source code, you can raise or lower these limits whenever you want.

Although the ability to record and play back keyboard macros will be sufficient for some applications, in many cases you will also need a way to edit these macros after they've been defined. For this reason, **Object Professional** provides a related unit, OPMACED, that implements a full-featured macro editor. This specialized editor allows you to easily insert, delete, and change the keys in a macro, which are represented as strings of one or more characters. The Escape key, for example, is displayed as "<Esc>", and the Enter key as "<Enter>", so you don't need to know anything about scan codes in order to use it.

For a comprehensive example of how to use **Object Professional**'s keyboard macros and macro editor, see the demonstration program SMACS.PAS. SMACS is a memory resident macro processor that includes most of the features found in commercial programs such as Borland's SuperKey. It comes with a file of sample macros, TURBO5.MAC, that you may find handy while using the Turbo Pascal editor.

The final unit in this group, OPREPLAY, implements a subset of the features found in OPMACRO. OPREPLAY does only one thing: execute (or "replay") pre-recorded macros on demand. It cannot record macros, or attach names to them, nor can it play them back automatically when a user presses a particular key. Obviously it is less powerful than OPMACRO, but there are cases where OPREPLAY is the better, if not the only, choice. For example, if you are writing a TSR that just needs to send a long series of specific keystrokes to the underlying application, you could use OPMACRO, but you'd be wasting valuable code and data space by using it when the smaller OPREPLAY would do. More importantly, if your TSR was written with OPSWAP you'd have no choice, because OPMACRO's interrupt $16 handler depends on data that is stored in the data segment and on the heap. OPREPLAY keeps all of its data in its own code segment, allowing it to be used in applications where OPMACRO can't, such as POPHELP.

B. OPMACRO: Keyboard Macros

This section describes all of the types, variables, constants, procedures and functions found in the interface section of the OPMACRO unit.

Constants

```
EndOfMacro = $FFFF;
```

Used to mark the end of a macro. Also used to mark unused entries in the DefinedKeys array.

```
Int16Handle = 12;
```

OPINT handle number assigned to the INT $16 handler.

```
MacroFileID  : LongInt = $04001087;
```

Macro file identification values. These values are stored at the beginning of any valid macro file to insure its validity.

```
MaxKeysInMacro = 1000;
```

The maximum number of keys in a macro. If you want to increase this limit, please keep in mind that a MacroRec contains an array[1..MaxKeysInMacro] of Word (see below), and that both OPMACRO and OPMACED declare a global variable of type MacroRec. In other words, the absolute limit for this constant is slightly over 32000, or 16000 if OPMACED is used. The practical limit is much lower.

```
MaxMacros = 100;
```

The maximum number of macros that OPMACRO can handle at one time.

```
PlaybackDelay : Word = 0;
```

Number of milliseconds to delay before playing back a particular keystroke in a macro.

Types

```
MacroRec = record
             NumKeys : Word;
             KeyArray : array[1..MaxKeysInMacro] of Word;
             Overflow : Word;
           end;
```

A macro. NumKeys is the number of keys in the macro, not counting the last, which is always EndOfMacro ($FFFF). KeyArray contains the actual keystrokes. Overflow holds the end of macro marker when KeyArray is completely full.

```
MacroRecPtr = ^MacroRec;
```

Pointer to a macro record.

```
StringPointer = ^String;
```

Pointer to a string.

Variables

```
DefinedKeys : array[1..MaxMacros] of Word;
```

Table of defined macro keys. Unused entries contain EndOfMacro.

```
MacroInProgress : Boolean;
```

If True, a macro is currently being played back.

```
MacroNames : array[1..MaxMacros] of Pointer;
```

Array of pointers to macro names.

```
MacroPointers : array[1..MaxMacros] of Pointer;
```

Array of pointers to defined macros.

```
MacroRecording : Boolean;
```

If True, macro recording is enabled.

```
MacrosAreOn : Boolean;
```
If True, macro processing is enabled.
```
ScrapMacro : MacroRec;
```
Used by macro engine when recording keystrokes.
```
ScrapMacroKey : Word;
```
Can be used to store the key associated with ScrapMacro.
```
ScrapMacroName : string;
```
Can be used to store the name associated with ScrapMacro.

Declaration

```
function AllocateMacro(Key : Word; var Macro; var Name : string) : Byte;
```

Purpose

Allocates a macro on the heap.

Description

This routine defines a macro and allocates space for both the macro and its name (if any). If there is already a macro assigned to Key, both the old macro and the old name are deleted and their memory deallocated. AllocateMacro returns one of the following values:

```
0  Macro allocated
1  Macro table full
2  Out of heap space
3  Macro was blank
```

Note that a return value of 3 is not necessarily an error code.

Example

```
var
  TempMacroKey : Word;
  TempMacro : MacroRec;
const
  NullString : string[1] = '';
begin
  {--get TempMacroKey and TempMacro--}
  case AllocateMacro(TempMacroKey, TempMacro, NullString) of
    {0 or 3 are OK, 1 and 2 are errors}
     1 : WriteLn('Macro table full');
     2 : WriteLn('Out of memory');
    else WriteLn('Macro saved');
  end;
end;
```

See Also

DeallocateMacro DefineMacro

Declaration

```
function AssignMacroName(Key : Word; var Name : string) : Boolean;
```

Purpose

Assign a Name to a macro.

Description

This routine allows you to change a macro name or assign a name to a macro that doesn't yet have one. It deletes the existing name assigned to the macro, if any, and returns False if there is insufficient memory for Name or if the macro associated with Key is not found.

Example

```
var Name : string;
begin
  Write('Enter name for macro: ');
  ReadLn(Name);
  if not AssignMacroName(ScrapMacroKey, Name) then
    WriteLn('Insufficient memory or macro not found');
end;
```

Assign a new name to the macro associated with ScrapMacroKey.

CharToMacro / ClearMacros / DeallocateMacro

Declaration

```
function CharToMacro(MacroChar : Char) : Word;
```

Purpose

Converts a character to a keyboard code/char representation.

Description

This function allows you to easily convert ASCII characters to key codes that can be inserted into a macro. If you need to convert an entire string into a macro, use StringToMacro instead.

Example

```
CharToMacro('A')
```

Returns $1E41

```
CharToMacro(^M)
```

Returns $1C0D

See Also

KeyToString StringToMacro

Declaration

```
procedure ClearMacros;
```

Purpose

Clears out the current macros.

Description

ClearMacros irreversibly deletes and deallocates all macros and macro names.

See Also

DefineMacro

Declaration

```
function DeallocateMacro(Key : Word) : Boolean;
```

Purpose

If the macro exists, this routine erases it and deallocates its memory.

Description

This routine is the counterpart to AllocateMacro. If Key is associated with a macro, it deletes both the macro and the corresponding macro name (if any), and deallocates their memory. DeallocateMacro returns False if Key is not associated with a macro.

Example

```
if not DeallocateMacro(ScrapMacroKey) then {error} ;
```

Get rid of any existing macro associated with ScrapMacroKey.

See Also

AllocateMacro DefineMacro

Declaration

```
function DefineMacro(Key : Word; var Macro) : Boolean;
```

Purpose

Define a new macro or delete an existing one.

Description

This low level routine lets you execute the main functions of AllocateMacro and DeallocateMacro, without their automatic allocation/deallocation of dynamic memory. It is intended primarily for internal use.

To define a macro: Key is the key to associate with Macro. DefineMacro will return False if the table is full or the macro is blank.

To delete a macro: Key is the key to delete and Macro must be a blank macro (NumKeys = 0). DefineMacro will return False if the macro was not found.

See the AllocateMacro and DeallocateMacro routines in OPMACRO.PAS for examples.

See Also

AllocateMacro DeallocateMacro

Declaration

```
function FindMacroIndex(Key : Word) : Word;
```

Purpose

Find the array index for the macro assigned to Key.

Description

This routine returns the array index for the macro corresponding to Key, or 0 if not found. This index can in turn be used to access elements in the MacroPointers, MacroNames, and DefinedKeys arrays. It is intended primarily for internal use, but you may need it to create routines that cannot be written using the higher level routines in OPMACRO. You might also want to use it as a fast alternative to MacroAddress, when checking to see if a particular key has already been defined. SMACS does this in several places, for example.

Example

```
Index := FindMacroIndex(MacroKey);
if Index <> 0 then WriteLn('Name: ', MacroNames[Index]^);
```

See Also

MacroAddress

InitMacros / KeyToString

Declaration

```
function InitMacros : Boolean;
```

Purpose

Initializes macro engine.

Description

InitMacros activates the macro engine. You might wish to deactivate it (using RemoveMacros) before executing a DOS shell, for example, then reactivate it with InitMacros. After a call to InitMacros, no macros are defined and both macro processing and macro recording are turned off. InitMacros returns False if INT $16 cannot be taken over. This can happen only if your program has already used the ISR handle used by the macro engine (Int16Handle). You do not need to call InitMacros at the start of your program, since that is done automatically.

Example

```
RemoveMacros;
Code := ExecDos('', True, NoExecDosProc);
if not InitMacros then {error};
```

See Also

RemoveMacros

Declaration

```
procedure KeyToString(Key : Word; var S : string; var Special : Boolean);
```

Purpose

Returns a string (S) representing a Key.

Description

This routine can be used, for example, when you want to display a list of defined macro keys. Special is False if a simple character (like <A>) is being returned, and True when a key with an extended scan code (like <F1>) is being represented.

Example

```
var I, Key : Word;
    S : string; Special : Boolean;
begin
  for I := 1 to MaxMacros do begin
    Key := DefinedKeys[I];
    if Key <> EndOfMacro then begin
      KeyToString(Key, S, Special); WriteLn(S);
    end;
  end;
end;
```

Displays a list of all keys that have macros attached to them.

See Also

CharToMacro

Declaration

```
function MacroAddress(Key : Word) : Pointer;
```

Purpose

Returns a pointer to the macro for a key, or nil if key is not defined.

Description

This routine allows you to determine whether or not a particular key has a macro associated with it. You might want to call this routine, for example, before calling StartMacroByKey or to see if you should ask the user to verify that an existing macro should be overwritten.

Example

```
var SavedKey : Word;
begin
  Write('Press key for macro to be played back later: ');
  MacrosOff;
  SavedKey := ReadKeyWord;
  MacrosOn;
  WriteLn;
  if MacroAddress(SavedKey) = nil then begin
    WriteLn('Macro not defined'); Exit;
  end;
  ...
  StartMacroByKey(SavedKey);
end;
```

See Also

StartMacroByKey

Declaration

```
function MacroCount : Word;
```

Purpose

Returns the number of macros that have been defined.

Description

The result is a number between 0 and MaxMacros (100).

Example

```
WriteLn(MacroCount, ' macros defined');
```

Declaration

```
procedure MacroRecordingOff;
```

Purpose

Turns macro recording off.

Description

Calling this routine simply tells the macro engine to stop saving keystrokes in ScrapMacro. It is your responsibility to assign this macro to a key and allocate space for it (using AllocateMacro, most likely).

Examples

See the PopupRecordingOff routine in SMACS.PAS for an example.

See Also

AllocateMacro MacroRecordingOn

MacroRecordingOn / MacrosOff / MacrosOn

Declaration
```
procedure MacroRecordingOn;
```

Purpose
Turns macro recording on.

Description
Calling this routine tells the macro engine to start saving all keystrokes entered by the user in ScrapMacro. It will continue saving keystrokes until recording is turned off with a call to MacroRecordingOff.

Examples
See the PopupRecordingOn routine in SMACS.PAS for an example.

See Also
MacroRecordingOff

Declaration
```
procedure MacrosOff;
```

Purpose
Turns macro processing off.

Description
This routine disables macro processing. SMACS calls this routine, for example, before prompting for the key associated with a macro to be edited. After the user has pressed a key, macro processing is reenabled with a call to MacrosOn.

Example
```
MacrosOff;
W := ReadKeyWord;
MacrosOn;
```
Turn macro processing off, read from the keyboard, turn it back on.

See Also
MacrosOn

Declaration
```
procedure MacrosOn;
```

Purpose
Turns macro processing on.

Description
This routine enables macro processing. Although some programs will simply need to call this routine once at the beginning of the program, in other cases you may need to temporarily turn macro processing off. SMACS, for example, disables macro processing before prompting for the key associated with a macro to be edited, then reenables it by calling this routine.

Example
```
MacrosOff;
W := ReadKeyWord;
MacrosOn;
```
Turn macro processing off, read from the keyboard, turn it back on.

See Also
MacrosOff

Declaration
```
function ReadMacroFile(FName : string; Merge : Boolean) : Byte;
```

Purpose
Read a macro file.

Description
This routine reads the contents of a macro file into memory. If Merge is False, any existing macros are deleted before the file is read. If True, existing macros that do not conflict with new ones are left in memory. ReadMacroFile returns 0 if successful, $FF if not a valid macro file, $FE if it ran out of memory, else IoResult.

Examples
```
if ReadMacros('MYMACROS.1', False) <> 0 then {error};
```
Existing macros overwritten.
```
if ReadMacros('MYMACROS.2', True) <> 0 then {error};
```
Existing macros not overwritten.

See Also
WriteMacroFile

Declaration
```
procedure RemoveMacros;
```

Purpose
Restores all vectors associated with macros.

Description
This routine disables the macro engine. You might wish to do this before executing a DOS shell, for example. The macro engine can be reactivated by calling InitMacros.

Example
```
RemoveMacros;
Code := ExecDos('', True, NoExecDosProc);
if not InitMacros then {error};
```

See Also
InitMacros StartMacro

SetEndOfMacroProc

Declaration

```
procedure SetEndOfMacroProc(P : Pointer);
```

Purpose

Set pointer to procedure to call when end of macro reached.

Description

SetEndOfMacroProc provides a hook that allows you to chain macros together. P is the address of a routine that is to be called immediately after a macro has been played back. When called, that routine may start a second macro (by calling StartMacro or StartMacroByKey), disable itself by calling SetEndOfMacroProc with nil as a parameter, or just do nothing.

When using this hook, a few important points must be kept in mind. First, the routine *must* be compiled under the FAR model, and it may not be nested inside another procedure. Second, since it is called from within OPMACRO's interrupt handler, it must use very little stack space. Third, it should not call any routine that might cause a reentrancy problem--i.e., it should not read from the keyboard or call any other routine that reads from the keyboard. It may, however, refer to global variables; OPMACRO's interrupt handler will insure that the DS register is initialized properly before it is called.

Example

```
var
  OurMacros : array[1..3] of MacroRecPtr;
  CurrentMacro : Byte;

  {$F+}
  procedure OurEndOfMacroProc;
  begin
    {decrement counter}
    Dec(CurrentMacro);
    {any macros left to play back?}
    if CurrentMacro = 0 then
      {disable the hook}
      SetEndOfMacroProc(nil)
    else
      {start the next macro}
      StartMacro(OurMacros[CurrentMacro]);
  end;
  {$F-}

begin
  {... init OurMacros with pointers to 3 macros, in reverse order ...}
  SetEndOfMacroProc(@OurEndOfMacroProc);
  {start macro #3}
  CurrentMacro := 3;
  StartMacro(OurMacros[CurrentMacro]);
end.
```

This code would play back three macros in succession.

See Also

StartMacro StartMacroByKey

Declaration
```
procedure StartMacro(Macro : Pointer);
```

Purpose
Starts execution of the specified macro.

Description
This routine lets you start any macro, even one that hasn't been officially defined (with DefineMacro or AllocateMacro). If a macro is already in progress it will immediately be replaced by the macro passed.

Examples
```
var MyMacroRec : MacroRec;
    MyMacroPtr : MacroRecPtr;
...
MyMacroPtr := @MyMacroRec;
StartMacro(MyMacroPtr);
```
Correct.
```
StartMacro(@MyMacroRec);
```
Also correct.

Declaration
```
procedure StartMacroByKey(Key : Word);
```

Purpose
Starts execution of the macro attached to key, if any.

Description
Unlike StartMacro, this routine allows you to play back a macro when all you know is the key associated with it. If a macro is already in progress, it will immediately be replaced by the specified macro.

Examples
```
const F1Key = $3B00;
...
StartMacroByKey(F1Key);
```
Play back the macro associated with <F1>, if any.

Declaration
```
procedure StringToMacro(S : string; MRP : MacroRecPtr; MaxKeys : Word);
```

Purpose
Initialize MRP^ from a string.

Description
This routine turns a string (S) into a macro. MRP should point to the macro record used to store the macro. MaxKeys is the maximum number of keys that MRP^ can hold.

Example
```
S := 'this is a test of StringToMacro'^M;
StringToMacro(S, @ScrapMacro, MaxKeysInMacro);
```
Converts S to a macro and stores it in ScrapMacro.

See Also
CharToMacro StringToScrapMacro

StringToScrapMacro / WriteMacroFile

Declaration

```
procedure StringToScrapMacro(S : string);
```

Purpose

Initialize ScrapMacro from a string.

Description

This routine forms a convenient "shell" around StringToMacro. It converts S to a macro and stores it in the global variable ScrapMacro. Note that this routine must not be used while macro recording is on, since ScrapMacro is used to store a macro that is being recorded.

Example

```
S := 'this is a test of StringToScrapMacro'^M;
StringToScrapMacro(S);
```

Converts S to a macro and stores it in ScrapMacro.

See Also

StringToMacro

Declaration

```
function WriteMacroFile(FName : string) : Byte;
```

Purpose

Write a macro file. Returns 0 if successful, else IoResult.

Description

This routine saves the current macros to a file. It returns 0 if successful, else IoResult.

Example

```
if WriteMacroFile('MYMACROS.3') <> 0 then {error};
```

See Also

ReadMacroFile

C. OPMACED: Macro Editor

This section describes all of the types, variables, constants, procedures and functions found in the interface section of the OPMACED unit.

Constants

```
MacEdCmd : Word;
```

Last command entered by the user in EditKeys.

```
MacEdCommands : CommandProcessor;
```

Command processor used by EditKeys. Note that this CommandProcessor provides the same hooks for background tasks, mouse support, and context-sensitive help that are described in the section about OPCMD.

```
MacEdKeyID : string[12] = 'opmaced keys';
```

ID string for use by installation programs. Appears just before MacEdKeySet.

```
MacEdKeyMax = 70;
```

Last available slot in MacEdKeySet.

```
MacEdKeySet : array[0..MacEdKeyMax] of Byte = (...);
```

Default key assignments for the macro editor.

```
NonMacroKey = $FFFF;
```

If this value is passed to EditKeys as the Key parameter, Key is not translated into a string and displayed while editing.

Variables

```
TempMacro : MacroRec;
```

Temporary macro used for editing. Since this macro is used only while editing, you may treat it as a scrap macro, as is done in SMACS.PAS.

EditKeys

Declaration

```
function EditKeys(Key : Word; var Macro : MacroRec;
                  MinCol, MinRow, MaxCol, MaxRow, Dim, Bright : Byte;
                  var Modified : Boolean) : MacroRecPtr;
```

Purpose

Edit the Macro associated with Key.

Description

EditKeys implements a full-featured macro editor. The editing window is defined by MinCol, MinRow, MaxCol, and MaxRow. The window should probably be at least 40 columns wide and five rows high, preferably more. Since the window scrolls vertically, you don't need to worry about macros that are too long to be displayed in their entirety.

The Dim and Bright parameters determine the video attributes used to draw the window border (Bright) and the keys in the macro. Normal keys are displayed in Dim; keys that must be represented with a string--<Esc>, <Enter>, <F1>, etc.--are displayed in Bright.

Note that the editing window used by EditKeys is not a Window in the OPWINDOW sense. The contents of the screen beneath the window are not saved, and the window may not have scroll bars or any of the other ornaments that can be added to a stackable window. Of course, EditKeys may be used inside of a stackable window to obtain these benefits.

Key is the key associated with the macro, and normally it is displayed at the top left corner of the window. If you don't want it displayed, pass NonMacroKey as the Key parameter.

Macro, a variable of type MacroRec, is never actually modified by EditKeys. EditKeys makes a copy of it, on which the actual editing is performed, and returns a pointer to the copy. If no edits were made to the Macro, the Modified parameter is set to False, otherwise to True.

The following commands are available in the macro editor:

<Enter>, <ClickLeft>
Stop editing.

<Esc>, <ClickRight>
Restore original macro and stop editing.

<Left>
Cursor left one key.

<Right>
Cursor right one key.

<Up>
Cursor up one line.

<Down>
Cursor down one line.

<Ins>
Toggle insert mode.

<Del>
Delete key at cursor.

<Bksp>
Delete previous key.

<CtrlBksp>
Delete all keys in macro.

<ScrollLock>
Toggle between command mode and literal mode. In literal mode, pressing any of the keys listed above will insert that key into the macro, rather than executing the command.

<F1>, <ClickBoth>
Help. Invoke help routine associated with MacEdCommands, if any.

Example

```
const
  NullString : string[1] = '';
var
  P, MyMacroPtr : MacroRecPtr;
  Modified : Boolean;
begin
  MacrosOff;
  Write('Macro key: ');
  ScrapMacroKey := ReadKeyWord;
  MacrosOn;

  {look for existing macro}
  P := MacroAddress(ScrapMacroKey);
  if P = nil then begin
    {macro not found--create blank macro to edit}
    ScrapMacro.NumKeys := 0;
    ScrapMacro.KeyArray[1] := EndOfMacro;
    P := @ScrapMacro;
  end;

  {edit the macro}
  MyMacroPtr := EditKeys(
    ScrapMacroKey, P^, 1, 1, ScreenWidth, ScreenHeight, $07, $0F, Modified);
  ClrScr;

  {if edits were made, allocate the macro (gets rid of old one)}
  if Modified then
    case AllocateMacro(ScrapMacroKey, MyMacroPtr^, NullString) of
      1..2 : WriteLn('Out of memory');
    end;
end;
```

This example demonstrates how to set up a full screen macro editor using EditKeys. For another example, see the EditMacro routine in SMACS.PAS.

D. OPREPLAY: Macro Playback

This section describes all of the types, variables, constants, procedures and functions found in the interface section of the OPREPLAY unit.

Constants

```
EndOfMacro = $0FFFF;
```

Used to mark the end of a macro.

```
MaxKeysInMacro = 1000;
```

The maximum number of keys in a macro. This value is completely arbitrary, and there is no need to adjust it.

Types

```
MacroRec = record
             NumKeys : Word;
             KeyArray : array[1..MaxKeysInMacro] of Word;
             Overflow : Word;
           end;
```

A macro. NumKeys is the number of keys in the macro, not counting the last, which is always EndOfMacro ($FFFF). KeyArray contains the actual keystrokes. Overflow holds the end of macro marker when KeyArray is completely full.

```
MacroRecPtr = ^MacroRec;
```

Pointer to a macro record.

```
ScrapMacroPtr : MacroRecPtr = nil;
```

If InitScrapMacroPtr has been called, this variable points to a "scrap" macro stored in OPREPLAY's code segment. It is meant to provide a convenient place to store small macros. With the exception of the StringToScrapMacro routine, OPREPLAY never uses it explicitly. The maximum size of the scrap macro is stored in ScrapMacroSize.

```
ScrapMacroSize : Word = 0;
```

Maximum number of keys in ScrapMacroPtr^. After a call to InitScrapMacroPtr, its value will be 127 unless OPSCRAP.ASM has been modified and reassembled.

Variables

```
CurrentMacroPtr : ^Pointer;
```

Pointer to address of current macro.

```
MacroInProgressPtr : ^Boolean;
```

Points to a Boolean that indicates whether a macro is currently being played back.

Declaration

```
function CharToMacro(MacroChar : Char) : Word;
```

Purpose

Converts a character to a keyboard code/char representation.

Description

This function allows you to easily convert ASCII characters to key codes that can be inserted into a macro. If you need to convert an entire string into a macro, use StringToMacro instead.

Examples

```
CharToMacro('A')
```

Returns $1E41.

```
CharToMacro(^M)
```

Returns $1C0D.

See Also

CharToMacro (OPMACRO)
StringToMacro
StringToScrapMacro

Declaration

```
procedure InitScrapMacroPtr;
```

Purpose

Initialize ScrapMacroPtr.

Description

This routine initializes ScrapMacroPtr to point to a macro declared in OPREPLAY's code segment that can hold up to 127 keys, and it sets ScrapMacroSize to 127. If this routine is not called, then you must manually initialize these two variables to point to a region of memory set aside for scrap macros before using ScrapMacroPtr explicitly or calling StringToScrapMacro.

Example

See the example for StringToScrapMacro.

Int16

Declaration

```
procedure Int16;
```

Purpose

The macro engine used to replay keystrokes.

Description

This is the interrupt handler that processes keyboard macros. It is your responsibility to activate this interrupt handler at the beginning of your program, and to deactivate it at the end.

Example

```
uses
  Dos, OpReplay;
const
  SaveInt16 : Pointer = nil;
var
  SaveExitProc : Pointer;

  {$F+}
  procedure OurExitProc;
  begin
    ExitProc := SaveExitProc;
    if SaveInt16 <> nil then
      SetIntVec($16, SaveInt16);
  end;
  {$F-}
begin
  SaveExitProc := ExitProc;
  ExitProc := @OurExitProc;
  GetIntVec($16, SaveInt16);
  SetIntVec($16, @Int16);
  ...
  SetIntVec($16, SaveInt16);
end;
```

Activate OPREPLAY's interrupt handler at the beginning of a program, and insure that it is deactivated at the end by installing a user exit handler.

Declaration

```
procedure SetEndOfMacroProc(P : Pointer);
```

Purpose

Set pointer to procedure to call when end of macro reached.

Description

SetEndOfMacroProc provides a hook that allows you to chain macros together. P is the address of a routine that is to be called immediately after a macro has been played back. When called, that routine may start a second macro (by calling StartMacro or StartMacroByKey), disable itself by calling SetEndOfMacroProc with nil as a parameter, or just do nothing.

When using this hook, a few important points must be kept in mind. First, the routine *must* be compiled under the FAR model, and it may not be nested inside another procedure. Second, since it is called from within OPREPLAY's interrupt handler, it must use very little stack space. Third, it should not call any routine that might cause a reentrancy problem–i.e., it should not read from the keyboard or call any other routine that reads from the keyboard. It may, however, refer to global variables; OPREPLAY's interrupt handler will insure that the DS register is initialized properly before it is called.

Most importantly, if OPREPLAY is being used in a swappable TSR, you must make sure that your end-of-macro routine is in a code segment that is always resident, and it may not refer to any variables in the data segment or on the heap.

Example

See the entry for the SetEndOfMacroProc routine in OPMACRO.

Declaration

```
procedure StartMacro(Macro : MacroRecPtr);
```

Purpose

Starts execution of the specified macro.

Description

This routine lets you start a macro that you've created by passing a pointer to it. If a macro is already in progress it will immediately be replaced by the macro passed.

Example

```
InitScrapMacroPtr;
StringToScrapMacro('this is a test of StartMacro'^M);
StartMacro(ScrapMacroPtr);
ReadLn(S);
```

Converts 'this is a test of StartMacro'^M to a macro, stored in ScrapMacroPtr^, then reads it into S.

StringToMacro / StringToScrapMacro

Declaration

```
procedure StringToMacro(S : string; MRP : MacroRecPtr; MaxKeys : Word);
```

Purpose

Initialize MRP^ from a string.

Description

This routine turns a string (S) into a macro. MRP should point to the macro record used to store the macro. MaxKeys is the maximum number of keys that MRP^ can hold.

Example

```
InitScrapMacroPtr;
S := 'this is a test of StringToMacro'^M;
StringToMacro(S, ScrapMacroPtr, ScrapMacroSize);
```

Converts S to a macro and stores it in ScrapMacro.

See Also

CharToMacro InitScrapMacroPtr

Declaration

```
procedure StringToScrapMacro(S : string);
```

Purpose

Initialize ScrapMacroPtr^ from a string.

Description

This routine forms a convenient "shell" around StringToMacro. It converts S to a macro and stores it in macro pointed to by ScrapMacroPtr^.

Note that ScrapMacroPtr *must* be initialized--either by a call to InitScrapMacroPtr or by setting it to point to your own scrap macro area--before calling this routine.

Example

```
InitScrapMacroPtr;
S := 'this is a test of StringToScrapMacro'^M;
StringToScrapMacro(S);
```

Converts S to a macro and stores it in ScrapMacroPtr^.

See Also

InitScrapMacroPtr

E. Sample Programs

There are two sample programs that illustrate the use of the units covered in this chapter. SMACS, documented below, demonstrates the use of OPMACRO and OPMACED. POPHELP, documented in Chapter 1, demonstrates the use of OPREPLAY.

SMACS: Macro Processor

SMACS is a memory resident keyboard macro processor much like Borland's SuperKey. To load SMACS, enter

```
SMACS [MacroFile]
```

at the DOS prompt, where the optional MacroFile parameter is the name of a file containing previously recorded macros. SMACS comes with a sample macro file, TURBO5.MAC, which you might want to try using in conjunction with the Turbo Pascal editor.

Once SMACS is loaded, you can pop it up at almost any time by pressing <Alt><~>. When you do, you'll see a menu that looks something like this:

```
──────── SUPER MACS: Macro Processor 1.00 ────────
Edit Delete Read Write List Status Configure Unload cApture Print
```

The following commands are available from the main menu:

Edit
Edit an existing macro or define a new one. When you issue this command, you will be asked first to press the key associated with the macro, and then for a name (optional) for the macro. Inside the macro editor, you can scroll through existing keystrokes using the cursor keys. By default the macro editor is in insert mode, so once the cursor has been positioned, you can insert new keys simply by pressing them. The <Ins> key toggles insert mode on and off. <Del> and <BkSp> delete macro keys. If you need to insert one or more keys that has a special function inside the macro editor--<Esc>, <Enter>, <Ins>, <Del>, the cursor keys, etc.--press <ScrollLock> to activate Literal mode, then press the key(s). To return to Command mode, press <ScrollLock> again. You can finish an editing session either by pressing <Enter>, which will save your changes, or <Esc>, which will not. For more details about the macro editor, see the section on the OPMACED unit.

Delete
Delete a previously defined macro or delete all macros. You will first be asked if you want to delete all macros. If you respond by pressing <Y>, you will be asked to confirm your choice. If you press <N> or <Enter>, you will be asked to press the key for the macro to be deleted. You can the press the appropriate key, or <Ctrl@> to cancel the operation.

Read
Read a macro file into memory. If there are any macros currently defined, you will be asked whether the current ones should be deleted before the new ones are read in. Note that you can also load a macro file from the DOS command line or from a batch file. For example,

```
cd \turbo
smacs turbo5.mac
turbo
```

would load the sample macros from TURBO5.MAC and delete any existing macros without asking for confirmation.

Write
Save the currently defined macros to a macro file. If the file already exists, you will be asked if it should be overwritten.

List
Displays a scrollable list of all currently defined macros. The <Up> and <Down> keys will scroll the list if there are more macros than can be displayed on the screen at once. <Home> and <End> take you to the beginning and end of the list, respectively. And <Esc> returns you to the main menu.

Status
Displays the number of macros that are currently defined and the amount of memory available for new ones.

Configure
Displays a menu that allows you to see and change the configuration of SMACS:

Main menu
Change the hotkey used to pop up SMACS's main menu.

Begin macro
Change the hotkey used to turn macro recording on.

End macro
Change the hotkey used to turn macro recording off.

Screen reader
Change the hotkey that activates the screen reader.

Macro memory
Change the amount of memory set aside for macros.

Playback delay
Specify the delay factor to use when playing back macros. If the delay factor is non-zero, SMACS will wait for the specified number of milliseconds before returning a particular key in the macro. Such delays are sometimes needed to prevent keystrokes from being lost. The default setting is 0; the maximum is 100.

Macro playback
Toggle macro processing on and off.

Remap keyboard
If this option is on, you can assign macros to keys like <F11> (on an enhanced keyboard) and <AltUp> that normally cannot be associated with macros. See the documentation for the OPENHKBD unit for a complete list of the new keys that this option makes available.

Save configuration
Save the current settings permanently. SMACS.EXE must be in 1) the current directory, 2) the drive and directory it was loaded from originally (if you're using DOS 3.0 or higher), or 3) a directory on the DOS PATH.

Use the <Up> and <Down> cursor keys to select an option, and press <Enter> to select it. The first four choices let you change one of SMACS's hotkeys. There are some limitations on the hotkeys that you can assign in this fashion, but you may want to experiment with these commands anyway. Except for the Macro memory setting, any changes made in the configuration menu will go into effect immediately. Assuming that

you use the Save configuration command to save your changes, the new Macro memory setting will go into effect the next time that SMACS is loaded.

Unload
Allows you, in many cases, to unload SMACS from memory without rebooting the system. SMACS will not unload itself if any program loaded above it uses the same interrupt vectors, so generally you cannot unload it unless it is the last program in memory. You can also unload SMACS from the command line by entering

```
SMACS -U
```

at the DOS prompt.

cApture
Allows you to capture the entire screen beneath the SMACS menu and save it to a file.

Print
Submit a file to the DOS 3.x (or higher) concurrent print facility. If PRINT.COM is not resident, or you are using DOS 2.x, an error message will be displayed.

Other Hotkeys

Besides the commands available through the main menu, SMACS has three other commands that are available through hotkeys (shown in the second column).

Begin macro **<Alt><=>**
Turn macro recording on. You will be prompted for the key to assign the macro to. If a macro has already been assigned to the key, you will be asked if the existing macro should be overwritten. Also, if you specify a key like <A> or <+> that is used for normal typing, you will be asked to confirm your choice of keys.

End macro **<Alt><->**
Turn macro recording off. You will be asked for a name to assign to the macro; if you don't wish to assign it a name, simply press <Enter>.

Screen reader **<Alt><Space>**
Activate the screen reader. The screen reader allows you to capture a portion of the current screen (a "block" of up to 1000 characters) and store it in a macro. Use the cursor keys to position the cursor, then press <Enter> to mark the start of the block of characters. Then move the cursor to the end of the block and press <Enter> a second time to select it. As you move the cursor, the screen will be "painted" in reverse video to indicate the current boundaries of the block. Once the block has been marked and selected, you will be asked for the key to assign the macro to. To cancel the operation before a block has been selected, press <Esc>.

Note that only 1 of the 4 commands available through hotkeys may be used at a time. You cannot, for example, activate the screen reader from inside the main menu.

14. Miscellaneous

A. Overview

This chapter contains the documentation for all the units that do not fall into any of our general categories. OPBCD provides a set of routines for performing high precision real arithmetic. OPCLONE assists in the writing of self-modifying programs and installation programs. OPERRHAN provides a facility that allows you to recover from most runtime errors. OPINLINE contains a collection of handy inline macros, as well as some routines for performing pointer arithmetic. And OPSORT gives you a speedy, non-recursive quicksort that works with almost any data type.

B. OPBCD: High-precision Reals

There are times when exceptional precision is required for real number calculations. Obvious examples are financial calculations, where a billion dollar sum must be accurate to the penny, or an amortization table that must retain at least 10 accurate digits for 30 years.

Turbo Pascal's 6-byte real data type does not provide sufficient accuracy for these situations. Two problems arise: first, there just isn't space in 6 bytes to retain sufficient precision; and second, because storage is in binary format, roundoff errors occur when convertng numbers to and from decimal format. Although 8087 data types supporting up to 19 significant digits are available, these reals still use a binary format, leading to roundoff errors.

Turbo Pascal version 3.0 provided a special version of the compiler just to solve this problem. This compiler defined a real type with 10 byte BCD (Binary Coded Decimal) format, providing 18 digits of accuracy and a range of 1E-63 to 1E+63. Later versions of Turbo Pascal have not provided support for BCD arithmetic, however, so **Object Professional** does.

The OPBCD unit offers an extended set of BCD arithmetic routines. A BCD real is represented as an array[1..10] of byte, with the same internal format as Turbo Pascal 3.0's BCD reals. Besides a full complement of arithmetic, comparison and format conversion routines, OPBCD provides the previously missing group of transcendental functions: Sqrt, Exp, Ln, Sin, Cos, and ArcTan. In addition, OPBCD includes a Form function for easy formatting of numeric results.

Except for the Form function, all the OPBCD routines are coded in assembly language for best speed and minimum code size. If no transcendental routines are used, OPBCD will add about 3K to a program's code size. If any transcendental routine is called, an additional 3K of code will be pulled in by the linker.

The non-transcendental functions retain full 18-digit accuracy for all operations. The transcendental operations retain at least 17 digits of accuracy, and are typically within one digit in the 18th place.

Because BCD calculations are not infix operations, and because Pascal functions cannot return structured types (such as arrays), the process of computing with BCD reals is now a bit more cumbersome. All OPBCD routines that compute BCD real results return them as var parameters. See the following section for detailed calling syntax.

Runtime errors in OPBCD are handled in a straightforward fashion. Whenever any numeric error occurs, whether it be divide by zero, floating point overflow, or an illegal operation such as the square root of a negative number, OPBCD generates a divide by zero interrupt (INT 0). By default, the Turbo Pascal system error handler takes this interrupt, reports a "Divide by zero," and shuts down the program. The error address reported by Turbo Pascal will be somewhere within the OPBCD unit, making it difficult to determine the true source of the problem. Optionally, you may use the OPERRHAN unit, in which case all OPBCD numeric errors are recoverable. By polling ExitCode after error recovery, you can make a more accurate determination of the true error location.

For an example of how to use the routines in OPBCD, see the source code for the OPCALC unit. DESKPOP uses OPCALC to implement a programmer's calculator that can paste the results of a calculation into the file currently loaded in the editor window. These values can be in the form of a regular number ('123.456') or in the form of a typed constant that is ideal for specifying BCD constants in your source code: '($42,$00,$00,$00,$00,$00,$00,$56,$34,$12)'

(= 123.456). OPCALC, we might add, demonstrates a useful technique for trapping arithmetic errors reported by the routines in OPBCD.

Special thanks to Philip R. Burns for his aid with the algorithms used in the transcendental functions.

Constants

```
BCDsize = 10;
```

Specifies the number of bytes used to store a BCD real. The value 10 is consistent with the BCD reals used by Turbo Pascal 3.0. By changing this constant and an equivalent constant in OPBCDLOW.ASM, BCD reals of higher accuracy can be obtained for all but the transcendental operations. (Important note: The value can be increased, but not decreased, and it *must* be an even number. You will need Borland's Turbo Assembler or MicroSoft's Macro Assembler in order to reassemble OPBCD.ASM and BCDTRANS.ASM.) Ten byte BCD reals offer 18 digits of accuracy, and two additional digits of accuracy would be gained for each extra byte used. Transcendental operations do not necessarily gain in accuracy because the algorithms to compute the functions are designed to provide only 18 digits of precision. More accurate algorithms would require more time and space than the existing ones.

```
CommaForPeriod  : Boolean = False;
```

If True, the Form function will return '.' for ',' and vice versa. This makes it a bit easier to write programs that can be used both in the U.S. and in countries that use commas for decimal points.

```
MaxBCD : BCD = ($7F,$00,$00,$00,$00,$00,$00,$00,$99); {+9.9e+63}
```

Maximum value for a BCD real.

```
MinBCD : BCD = ($FF,$00,$00,$00,$00,$00,$00,$00,$99); {-9.9e+63}
```

Minimum value for a BCD real.

```
MoneySign  : Char = '$';
```

This character is used when displaying values in dollars-and-cents format. If desired, you could change this constant to #156, for example, allowing Form to display amounts in terms of British pounds rather than dollars.

```
PiBcd  : BCD = ($40, $24, $93, $97, $58, $53, $26, $59, $41, $31);
```

Specifies Pi to 18 digits in BCD format.

```
ZeroBcd  : BCD = ($0, $0, $0, $0, $0, $0, $0, $0, $0, $0);
```

0 in BCD format.

Types

```
BCD  = array[1..BCDsize] of Byte;
```

The internal format of a BCD real. See OPBCD.ASM for a more detailed description of the BCD format.

AbsBCD / AddBCD / ArcTanBCD

Declaration

```
procedure AbsBCD(B1 : BCD; var B2 : BCD);
```

Purpose

Returns absolute value of B1 in B2.

Description

This is a very fast operation that requires clearing only one bit in the input real, and transferring ten bytes to the output.

Example

```
{Assume B1 holds -1.2345}
AbsBCD(B1, B2);
```

B2 returns 1.2345.

Declaration

```
procedure AddBCD(B1, B2 : BCD; var B3 : BCD);
```

Purpose

Add B1 to B2 and put result in B3.

Description

AddBCD creates a temporary result with 19 digits of accuracy, then rounds this back to 18 digits if required.

If the result exceeds 1.0E64, an overflow error occurs.

Example

```
{Assume that B1 holds 0.999999999999999999}
AddBCD(B1, B1, B2);
```

B2 then contains 2.000000000000000000 because the intermediate result contains 19 digits and is rounded up.

Declaration

```
procedure ArcTanBCD(B1 : BCD; var B2 : BCD);
```

Purpose

Returns arc tangent of B1 in B2.

Description

Like the equivalent operation with normal reals, ArcTanBCD returns the angle, in radians, whose tangent is B1. ArcTanBCD always returns values between -Pi/2 and +Pi/2. Any value within the range of BCD reals is acceptable as an input parameter.

Example

```
{Assume that B1 holds 1.0}
ArcTanBCD(B1, B2);
{Assume that B3 holds 4.0}
MultBCD(B2, B3, B4);
```

Then B4 returns 3.14159265358979323, which equals Pi to within one digit in the last place.

Declaration
```
function BCDtoReal(B : BCD) : Real;
```

Purpose
Convert a BCD to a real.

Description
This routine converts a BCD to a 6-byte non-8087 real. We assume that you wouldn't be using the OPBCD unit if you were already using 8087 reals.

The routine works by converting a BCD to the equivalent full-precision ASCII string (a fast operation), and then calling the SYSTEM unit Val procedure to convert the string to a real. The 18 digits of BCD accuracy are rounded to the 11 digits of Turbo accuracy.

See Also
LongIntToBCD RealToBCD

Declaration
```
procedure CosBCD(B1 : BCD; var B2 : BCD);
```

Purpose
Returns cosine of B1 in B2.

Description
Like Turbo's normal Cos function, CosBCD takes an angle expressed in radians and returns the cosine in parameter B2. The largest acceptable input value is 3.14E9 radians, and the smallest is -3.14E9. The output value is always in the range -1.0 to +1.0.

CosBCD is accurate within three digits in the 18th place for all cases where we've found a guaranteed precise reference. It also matches the results of an 80287 numeric coprocessor within three digits in the 18th place for a broad range of input values.

Example
```
CosBcd(PiBcd, B1);
```
Returns in B1 the value -0.999999999999999997 (should be -1.0).

See Also
SinBcd

Declaration
```
procedure DivBCD(B1, B2 : BCD; var B3 : BCD);
```

Purpose
Divide B1 by B2 and put result in B3.

Description
The division algorithm used is analogous to the "subtract and shift" technique taught in grade school. It is precise for all 18 digits.

If the result exceeds 1.0E64, an overflow error occurs. If the result is less than 1.0E-63, zero is returned.

See Also
MultBCD

EqualBCD / ExpBCD

Declaration

```
function EqualBCD(B1, B2 : BCD) : Boolean;
```

Purpose

Returns True if B1 = B2.

Description

EqualBcd compares the two operands for exact equality. The smallest difference will cause it to return False.

To compare for approximate equality, you should define a function like the following:

```
function ApproxEqual(B1, B2, Epsilon : BCD) : Boolean;
  {-Return true if Abs(B1-B2) < Epsilon}
var
  Difference : BCD;
begin
  SubBCD(B1, B2, Difference);
  AbsBCD(Difference, Difference);
  ApproxEqual := LessBCD(Difference, Epsilon);
end;
```

To see if two items matched within one tenth of a cent, you would make a call like:

```
RealToBcd(0.001, OneTenthCent);
if ApproxEqual(B1, B2, OneTenthCent) then
  ...;
```

See Also

NotEqualBCD

Declaration

```
procedure ExpBCD(B1 : BCD; var B2 : BCD);
```

Purpose

Returns the exponential of B1 in B2.

Description

The largest acceptable value of B1 is 147.36. Values larger than this will cause an overflow error.

The smallest value of B1 for which the result retains any resolution is -145.06. ExpBCD returns 0.0 for values smaller than this.

ExpBCD is accurate within 1 digit in the 18th place for all cases where we've found a guaranteed precise reference. It matches the results of an 80287 coprocessor within 3 digits in the 18th place across a broad range of inputs.

Example

```
{Assume B1 contains 1.0}
ExpBCD(B1, B2);
```

Returns in B2 the value 2.71828182845904524 (e accurate to 18 digits).

See Also

LnBcd

Declaration
```
function Form(Mask : string; B : BCD) : string;
```

Purpose
Returns a formatted string with digits from B merged into the Mask.

Description
Form provides the most important features of the Turbo 3.0 BCD Form function. This function offers advanced formatting capabilities for real numbers. It is the only OPBCD function written in Pascal--if you wish, it should be easy to modify or add features.

See the description of the Form function in the OPSTRING section for complete information.

See Also
Form (OPSTRING) StrExpBCD
StrBCD

Declaration
```
procedure FracBCD(B1 : BCD; var B2 : BCD);
```

Purpose
Returns the fractional part of B1 in B2.

Description
FracBCD's result is the same as computing FracBCD := B1-IntBCD(B1).

Examples
```
{Assume B1 contains 1.46}
FracBCD(B1, B2);
```
Returns 0.46 in B2.
```
{Assume B1 contains -1.46}
FracBCD(B1, B2);
```
Returns -0.46 in B2.

Declaration
```
function GreaterBCD(B1, B2 : BCD) : Boolean;
```

Purpose
Returns True if B1 > B2.

Description
All of the BCD comparison functions share a common low level routine that does the actual comparison and returns the 8086 flags set for the result. The top level function then interrogates the flags to return an appropriate boolean value.

The comparison works with all 18 digits. Thus 1.00000000000000001 is greater than 1.00000000000000000.

GreaterEqualBCD / IntBCD / LessBCD

Declaration

```
function GreaterEqualBCD(B1, B2 : BCD) : Boolean;
```

Purpose

Returns True if B1 > = B2.

Description

All of the BCD comparison functions share a common low level routine that does the actual comparison and returns the 8086 flags set for the result. The top level function then interrogates the flags to return an appropriate boolean value.

The comparison works with all 18 digits.

Declaration

```
procedure IntBCD(B1 : BCD; var B2 : BCD);
```

Purpose

Returns the integer part of B1 in B2.

Description

If B1 is greater than or equal to zero, then IntBcd returns the greatest integer less than or equal to B1. If B1 is less than zero, then IntBCD returns the smallest integer greater than or equal to B1.

Note that the result is a BCD real with no fractional part, not an actual integer.

Examples

```
{Assume B1 contains 1.46}
IntBCD(B1, B2);
```

Returns 1.0 in B2.

```
{Assume B1 contains -1.46}
IntBCD(B1, B2);
```

Returns -1.0 in B2.

Declaration

```
function LessBCD(B1, B2 : BCD) : Boolean;
```

Purpose

Returns True if B1 < B2.

Description

All of the BCD comparison functions share a common low level routine that does the actual comparison and returns the 8086 flags set for the result. The top level function then interrogates the flags to return an appropriate boolean value.

The comparison works with all 18 digits.

Declaration
```
function LessEqualBCD(B1, B2 : BCD) : Boolean;
```

Purpose
Returns True if B1 < = B2.

Description
All of the BCD comparison functions share a common low level routine thatwhich does the actual comparison and returns the 8086 flags set for the result. The top level function then interrogates the flags to return an appropriate boolean value.

The comparison works with all 18 digits.

Declaration
```
procedure LnBCD(B1 : BCD; var B2 : BCD);
```

Purpose
Returns the natural logarithm of B1 in B2.

Description
The input to LnBcd must be a positive number or a numeric error will occur.

LnBCD is accurate within one digit in the 18th place for all cases where we've found a guaranteed precise reference. It matches the results of an 80287 numeric coprocessor within two digits in the 18th place for a broad range of inputs.

Examples
```
{Assume B1 contains 2.0}
LnBCD(B1, B2);
```
Returns 6.93147180559945308E-01 in B2 (last digit should be 9).

See Also
ExpBCD

Declaration
```
procedure LongIntToBCD(L : LongInt; var B : BCD);
```

Purpose
Convert a LongInt to a BCD.

Description
This procedure provides a convenient method to convert integer constants to BCD values. The BCD holds all 10 digits of the largest LongInt exactly.

Examples
```
LongIntToBCD(0, B1);
```
Puts the value 0 in B1.
```
LongIntToBCD(MaxLongInt, B1)
```
Puts 2147483647 in B1.

See Also
RealToBCD

MultBCD / NotEqualBCD

Declaration

```
procedure MultBCD(B1, B2 : BCD; var B3 : BCD);
```

Purpose

Multiply B1 by B2 and put result in B3.

Description

The multiplication algorithm used is analogous to the "multiply digits and carry" technique taught in grade school. It is precise for all 18 digits.

If the result exceeds 1.0E64, an overflow error occurs. If the result is less than 1.0E-63, zero is returned.

See Also

DivBCD

Declaration

```
function NotEqualBCD(B1, B2 : BCD) : Boolean;
```

Purpose

Returns True if B1 < > B2.

Description

NotEqualBcd returns True unless the inputs agree in all 18 digits. Use caution in applying such comparisons.

To allow for approximate equality, you should define a function like the following:

```
function NotApproxEqual(B1, B2, Epsilon : BCD) : Boolean;
  {-Return true if Abs(B1-B2) > Epsilon}
var
  Difference : BCD;
begin
  SubBCD(B1, B2, Difference);
  AbsBCD(Difference, Difference);
  ApproxEqual := GreaterBCD(Difference, Epsilon);
end;
```

See Also

EqualBCD

Declaration

```
procedure RealToBCD(R : Real; var B : BCD);
```

Purpose

Convert a real to a BCD.

Description

This procedure provides a convenient method for initializing BCD real constants. As long as the precision of the constant does not exceed that of a 6-byte real, the BCD will receive an exact constant value.

If you have constants that require more than 11 digits of accuracy, we suggest that you use the calculator in DESKPOP to generate typed constants in BCD array format. In this way, the constants can be accurate in all 18 digits. Alternatively, you may use the ValBCD procedure to convert a string representation of the constant to a BCD real.

RealToBCD works only to convert Turbo's 6-byte reals. It will not work with 8087 format reals. We assume that you wouldn't be using the OPBCD unit if you were already using 8087 reals.

Examples

```
RealToBCD(1.5, B1);
```

Initializes B1 to 1.5.

```
RealToBCD(-1.78234976E-22, B1);
```

Initializes B1 to -1.78234976E-22.

See Also

LongIntToBCD

Declaration

```
function RoundBCD(B1 : BCD) : LongInt;
```

Purpose

Returns the value of B1 rounded to the nearest long integer.

Description

If B1 exceeds MaxLongInt (2,147,483,647), then RoundBCD triggers an overflow error. Otherwise it performs normal rounding to generate a LongInt result.

If B1 is greater than or equal zero, then RoundBCD returns TruncBCD(B1+0.5). Otherwise, it returns TruncBCD(B1-0.5).

Examples

```
{Assume B1 contains 1.46}
RoundBCD(B1)
```

Returns 1.

```
{Assume B1 contains 1.50}
RoundBCD(B1)
```

Returns 2.

```
{Assume B1 contains -1.46}
RoundBCD(B1)
```

Returns -1.

```
{Assume B1 contains -1.50}
RoundBCD(B1)
```

Returns -2.

See Also

TruncBCD

SinBCD / SqrBCD / SqrtBCD

Declaration

```
procedure SinBCD(B1 : BCD; var B2 : BCD);
```

Purpose

Returns the sine of B1 in B2.

Description

Like Turbo's normal Sin function, SinBCD takes an angle expressed in radians and returns the sine in parameter B2. The largest acceptable input value is 3.14E9 radians, and the smallest is -3.14E9.

SinBCD is accurate within one digit in the 18th place for all cases where we've found a guaranteed precise reference. It also matches the results of an 80287 numeric coprocessor within one digit in the 18th place for a broad range of inputs.

Example

```
RealToBcd(4.0, B1);
DivBcd(PiBCD, B1, B2); {Compute Pi/4}
SinBcd(B2, B4);
```

Returns Sin(Pi/4) in B4 = 7.07106781186547525E-01.

See Also

CosBCD

Declaration

```
procedure SqrBCD(B1 : BCD; var B2 : BCD);
```

Purpose

Returns the square of B1 in B2.

Description

This operation is equivalent to MultBCD(B1, B1, B2). It runs a bit faster and is slightly easier to call.

If the result exceeds 1.0E64, an overflow error occurs. If the result is less than 1.0E-63, zero is returned.

See Also

MultBCD

Declaration

```
procedure SqrtBCD(B1 : BCD; var B2 : BCD);
```

Purpose

Returns the square root of B1 in B2.

Description

The input must be non-negative or a numeric error occurs.

The algorithm used is Heron's approximation. It is accurate for all 18 digits and requires 4 iterations.

Example

```
{Assume B1 holds 2.0}
SqrtBCD(B1, B2);
```

Returns 1.41421356237309505, accurate to 18 digits.

Declaration

```
function StrBCD(B : BCD; Width, Places : Byte) : string;
```

Purpose

Return a BCD as a string.

Description

This routine is functionally equivalent to the Turbo Pascal syntax Str(R:Width:Places, S) StrBCD returns a string in floating point format (no exponent value is included). Note that StrBCD is a function rather than a procedure like Str.

If you wish to obtain a BCD real in exponential format, use the function StrExpBCD. The Form function provides yet another way to format BCD reals for output.

The maximum useful number of decimal Places to specify is 18, although StrBCD will return more (filled with zeros) if you ask for them. If the actual width of the string exceeds the requested Width, StrBcd will use the actual width. If the actual width is less, the answer is right justified in a string of the requested width.

Examples

```
WriteLn(StrBCD(B1, 10, 2));
```

Writes the BCD real B1 to the screen the same way that

```
WriteLn(R1:10:2);
```

would write it for a normal real.

```
StrBCD(B1, 0, 0)
```

Returns the real B1 rounded to the nearest integer in a string of minimum width.

See Also

Form

Declaration

```
function StrExpBCD(B : BCD; Width : Byte) : string;
```

Purpose

Return B as a string in exponential format.

Description

This routine formats a BCD real as an exponential format string. If Width is 0, the result is a 24-character string of the form -n.nnnnnnnnnnnnnnnnE+ee; the first character is a blank for non-negative numbers. If Width is not 0, the string returned will be Width characters long. If Width is less than 24, the excess digits will be rounded; if it is greater than 24, the value will be right justified in the string, which will be padded at the front with blanks. If Width is greater than 0 and less than 8, a Width of 8 will be used.

Example

```
{Suppose B1 has 123456.9}
StrExpBCD(B1, 0)
```

Returns ' 1.23456900000000000E+05'.

```
StrExpBCD(B1, 9)
```

Returns ' 1.23E+05'.

See Also

Form

SubBCD / TruncBCD

Declaration

```
procedure SubBCD(B1, B2 : BCD; var B3 : BCD);
```

Purpose

Subtract B2 from B1 and put result in B3.

Description

SubBCD creates a temporary result with 19 digits of accuracy, then rounds this back to 18 digits if required.

If the result exceeds 1.0E64, an overflow error occurs. If the result is less than 1.0E-63, zero is returned.

Declaration

```
function TruncBCD(B1 : BCD) : LongInt;
```

Purpose

Returns the greatest long integer less than or equal to B1.

Description

If B1 is greater than or equal to zero, TruncBCD returns the greatest integer less than or equal to B1. Otherwise, TruncBCD returns the smallest integer greater than or equal to B1.

If B1 exceeds MaxLongInt (2,147,483,647), an overflow error occurs.

Examples

```
{Assume B1 contains 1.46}
TruncBCD(B1)
```

Returns 1.

```
{Assume B1 contains 1.999999}
TruncBCD(B1)
```

Returns 1.

```
{Assume B1 contains -1.999999}
TruncBCD(B1)
```

Returns -1.

See Also

RoundBCD

Declaration

```
procedure ValBCD(S : string; var B : BCD; var Code : Word);
```

Purpose

Convert a string to a BCD.

Description

This procedure provides for BCD reals the equivalent of Turbo's Val routine. S is a string in any valid floating point or exponential format. Leading blanks are acceptable, but trailing blanks are not.

If the conversion is successful, Code will contain zero, and B will hold the converted real. Otherwise, Code will contain a non-zero value that points to the input string location causing the error, and B will contain an indeterminate value. If the string specifies a real too large to fit into BCD format, Code will point beyond the end of the string.

Examples

```
ValBCD('0.57721566490153286061', B1, Code);
```

Returns Code=0 and the value of Euler's constant loaded into B1. Note that excess digits are truncated to fit into the 18 digit mantissa of BCD reals.

```
ValBCD('  1.0E22  ', B1, Code);
```

Returns Code=9 because trailing blanks are not allowed.

```
ValBCD('-1.0E99', B1, Code);
```

Returns Code=8 because the specified value exceeds the range of BCD reals.

See Also

Form
StrBCD
StrExpBCD

C. OPCLONE: Program Customization

As software applications continue to grow in terms of both size and complexity, more and more programmers find it necessary to provide users a way to customize their programs to satisfy individual needs and desires. There are numerous ways to accomplish this task, but this is not the place to enumerate them or to weigh their merits. Our concern here is with two of the more common methods, both of which can be implemented with the aid of OPCLONE.

The first is the self-modifying program. If you are programming in Turbo Pascal, the key to writing such a program is to store all of your "configuration data" in the form of typed constants. These typed constants are typically stored in one or more groups, each preceded by a unique ID string. For example:

```
const
  {mark start of configuration data}
  MyProgIdString : string[9] = 'MYPROG ID';
  {configuration data}
  MyProgColors : ColorSet = (...);
  MyProgHotKeys : array[1..100] of Byte = (...);
  {mark end of configuration data}
  MyProgCfgEnd : Byte = 0;
```

By writing the current values of these typed constants to the appropriate location in the EXE file, determined by searching for the ID string, we can create a customized version of the program. (For historical reasons, this is often referred to, at least by Turbo Pascal programmers, as "cloning"--hence the name of the unit.) The next time the program is run, the saved values will be the defaults, and the user will not have to re-customize the program unless he wants to change something else.

The second is the installation (or configuration) program. It differs from the first method in that the program being configured is completely separate from the one performing the configuration. The main difference between the two methods is that the installation program must read the current values for the variables out of the other program's EXE file before allowing the user to modify them. A self-modifying program can skip this step. Otherwise, the two techniques are essentially identical:

1. Open the EXE file to be modified
2. Locate the configuration data within the EXE file by searching for the ID string
3. Read in the current values (installation program only)
4. Allow the user to modify the values (in a self-modifying program this has probably been done already)
5. Write the modified values back to disk, at the location found in step 2
6. Close the EXE file

There are numerous variations on this basic technique--some do not require an ID string, for example, and some can skip the second step after the operation has been performed once--but this version of it is the one most widely used, and it has proven over time to be both safe and reliable.

In OPCLONE, this technique is implemented in the form of an object called a Cloner. A Cloner is a quite simple object, with few methods for you to learn, making it very easy to write self-modifying programs:

```
{$A-}               {required for OPCLONE}
program CloneTest;  {EXCLONE.PAS}
uses
  Dos, OpInline, OpString, OpRoot, OpClone;
const
  {mark start of installation area}
  Id : string[12] = 'CloneTest ID';
  {configuration data}
  TimesRun : LongInt = 0;
  TestString : string[20] = 'Hello World!';
  {mark end of installation area}
  CfgEnd : Byte = 0;
var
  C : Cloner;

begin
  {show that it works}
  WriteLn('Program has been run successfully ', TimesRun, ' time(s)');
  Inc(TimesRun);
  WriteLn('Test string = "', TestString, '"');
  Write('Enter new test string: ');
  ReadLn(TestString);
  {initialize cloner}
  if not C.InitCustom('EXCLONE.EXE', UpdateAll, DefBufSize) then
    WriteLn('Unable to modify EXE file')
  else begin
    {locate configuration data}
    if not C.FindDefaultsEnd(Id, SizeOf(Id), 0) then
      WriteLn('Unable to locate configuration data')
    else begin
      {write out the new values}
      C.StoreDefaults(C.GetPos, Id, Ofs(CfgEnd)-Ofs(Id));
      {check for errors}
      if C.GetLastError <> 0 then
        WriteLn('Error storing configuration data')
      else
        WriteLn('Configuration data stored successfully');
    end;
    {close the EXE file}
    C.Done;
  end;
end.
```

The first thing you should notice here is the compiler directive {$A-} at the top. This directive insures that the compiler does not word-align any of the variables in the "installation area." Although it is technically required only in installation programs and the programs they modify, you'd be wise to add it anytime you use OPCLONE.

The next noteworthy point concerns the installation area. It begins with an ID string (Id) and ends with a dummy variable that marks the end of the installation area. In between are all the variables that we refer to collectively as the "configuration data." Although it is not necessary to add a dummy variable at the end, there are two good reasons for doing so: 1) it simplifies the calculation of the size of the installation area, and 2) it insures that you can add or delete items from your installation areas without having to remember to change any of your calls to Cloner's methods.

The first part of the example program simply demonstrates that the technique works. You can prove this to yourself if you wish by compiling the program (call it EXCLONE.PAS) and running it a couple of times.

The call to C.InitCustom opens the EXE file and allocates a 4K buffer (DefBufSize = 4096) that will be used when searching for the ID string. The call to C.FindDefaultsEnd performs the actual search. If the search succeeds, the offset into the EXE file for the start of the installation area can be retrieved by calling the GetPos function. In our example, this value is passed directly to C.StoreDefaults, which writes the current contents of the installation area back to the EXE file. The call to C.GetLastError checks to see if an error occurred when writing the data to disk. And, finally, the call to C.Done closes the EXE file, updates its date-time stamp, and deallocates the search buffer.

A Cloner can also, of course, be used to write an installation program. The main difference, as far as use of the Cloner is concerned, is that you have to call LoadDefaults to load the configuration data for the program being installed into the installation program's workspace. For an example of such a program, see the source code for DPINST, the installation program for DESKPOP. For a more comprehensive example of how to use OPCLONE to write a self-modifying program, see SMACS.PAS.

Constants

```
MinBufSize = 512;
```

Minimum buffer size.

```
DefBufSize = 4096;
```

Default buffer size.

Types

```
Cloner =
  object(Root)
    ...
  end;
```

Object used to assist in the writing of configuration programs.

```
DateUpdateType = (UpdateNone, UpdateDate, UpdateAll);
```

Type used to indicate what a Cloner object should do concerning the date-time stamp on the EXE file after updating it. UpdateNone indicates that the original date-time stamp should be maintained. UpdateDate indicates that the date should be changed, but not the time. UpdateAll indicates that both the date and the time should be changed, as would normally occur after a file has been modified.

```
SearchBuffer = array[0..DefBufSize-1] of Char;
```

Buffer used to locate a search string within an EXE file.

Declaration

```
destructor Done; virtual;
```

Purpose

Close EXE file, adjust time stamp, and deallocate search buffer.

Description

This method should be called when you are finished with the object. It does three things: 1) closes the EXE file being cloned; 2) adjusts the date-time stamp of the file based on the instructions given to the constructor; and 3) deallocates the search buffer allocated by the constructor.

Example

See the example at the beginning of this section.

See Also

Init InitAndFind

Declaration

```
function FindDefaultsEnd(var ID; IdSize : Word; Skip : LongInt) : Boolean;
```

Purpose

Find the ID in the EXE file, searching from the end backward.

Description

This method is used to find the position of an ID string within the EXE file. It begins its search at the end of the EXE file and continues searching backward until the string is found or the entire EXE file has been searched. It returns True if the ID string is found, and False if it isn't. If the ID string is found, its location within the file may be obtained by calling GetPos.

ID is the ID string to search for, and IdSize indicates its size. ID will typically be a regular Turbo string ('IdString, SizeOf(IdString)'), an array of characters ('IdArray, SizeOf(IdArray)'), or the first byte of a Turbo string ('IdString[1], Length(IdString)'). But any data type is valid as long as its value never changes and it is guaranteed to appear only once in the EXE file being configured.

Skip is the number of bytes to be skipped when the search begins. For example, if Skip = 1000, then the final 1000 bytes in the EXE file would not be searched. This value should typically be 0, although in some cases you might want to use a higher value if the ID string is known to appear far from the end of the file and performance is an important consideration.

If FindDefaultsEnd returns False, you can determine the cause by calling GetLastError. The error code will typically be either an I/O error code (likely due to a disk read error), or epNonFatal + ecStringNotFound, which means that the search string could not be found. Likely reasons for such an error include 1) programmer error (especially if Skip < > 0), 2) a corrupt EXE file, and 3) an EXE file that has been "packed" using a program such as MicroSoft's EXEPACK or the PACK utility in Turbo Analyst.

Example

See the example at the beginning of this section.

See Also

FindDefaultsStart GetPos
GetLastError LoadDefaults

FindDefaultsStart / GetLastError

Declaration

```
function FindDefaultsStart(var ID; IdSize : Word; Skip : LongInt) : Boolean;
```

Purpose

Find the ID in the EXE file, searching from the start forward.

Description

FindDefaultsStart works much like FindDefaultsEnd, except that it starts searching at the beginning of the EXE file, rather than at the end. FindDefaultsStart should generally be used only in cases where the search string (ID) is known to occur near the beginning of the EXE file, which will typically happen only if the configuration data was linked into the program as part of an OBJ file, not as a group of typed constants.

The parameters are identical to those for FindDefaultsEnd, except that Skip is the number of bytes to skip at the *beginning* of the EXE file.

Example

```
const
  Id : string[9] = 'MYPROG ID';
var
  C : Cloner;
begin
  if not C.Init('MYPROG.EXE', UpdateAll) then
    {handle error}
  else begin
    if not C.FindDefaultsStart(Id, SizeOf(Id), 0) then
      {handle error}
    else begin
      ...
    end;
    C.Done;
  end;
end;
```

See Also

FindDefaultsEnd LoadDefaults

Declaration

```
function GetLastError : Word;
```

Purpose

Get the code for the last error.

Description

This method returns the code for the last error that occurred in one of the other methods (excluding the constructors, which return error codes in InitStatus). The error code is interpreted as shown in Appendix D.

Note that, like IoResult, GetLastError resets the internal variable used to store the error code that it returns, so a second call to GetLastError will always return 0.

Example

See the example at the beginning of this section.

Declaration
```
function GetPos : LongInt;
```

Purpose
Get position where an ID string was found.

Description
This method may be called after a successful call to FindDefaultsStart or FindDefaultsEnd to determine the offset in the file where the ID string was found. If an error is pending, GetPos will always return 0 (never a valid offset for an ID string). Otherwise, it returns a value between 1 and FileSize(ExeFile)-1. This value may then be passed to LoadDefaults to read in the configuration data just located.

Note that the offset it returns is that of the *start* of the ID string. Hence, when calling LoadDefaults, the final parameter (Bytes) should include the size of the ID string.

Example
See the example at the beginning of this section.

See Also
FindDefaultsEnd
FindDefaultsStart
LoadDefaults

Declaration
```
constructor Init(FName : string; DUT : DateUpdateType);
```

Purpose
Open EXE file for cloning and allocate buffer.

Description
This constructor simply calls InitCustom with a buffer size of DefBufSize.

Declaration
```
constructor InitCustom(FName : string; DUT : DateUpdateType; BufSize : Word);
```

Purpose
Open EXE file for cloning and allocate buffer.

Description
This constructor attempts to open the EXE file specified by FName and to allocate the buffer used when searching the file for ID strings. BufSize is the size of the buffer to allocate. It should be in the range MinBufSize..65521.

InitCustom will fail only if the EXE file could not be opened or the search buffer could not be allocated. You can determine the precise cause of failure by checking the global variable InitStatus, declared in OPROOT.

DUT determines whether and how the date-time stamp for the EXE file will be adjusted when the EXE file is closed by the Done method. See the entry for DateUpdateType, above, for a discussion of the possible values for this parameter.

Example
See the example at the beginning of this section.

See Also
Done
InitAndFind

InitAndFind

Declaration

```
constructor InitAndFind(FName : string; DUT : DateUpdateType; var ID;
                        IdSize : Word);
```

Purpose

Initialize object and find ID.

Description

This constructor combines the functionality of Init and FindDefaultsEnd, and is provided primarily as a convenience. It will fail if 1) the EXE file (FName) could not be opened, 2) the search buffer could not be allocated, or 3) the ID string (ID) could not be found. You can determine the precise cause of failure by checking the global variable InitStatus, declared in OPROOT.

InitAndFind does little more than pass the first two parameters to Init, and the second two to FindDefaultsEnd, so for further details on how it works you should consult the documentation for those two methods. Note that InitAndFind always passes 0 as the final parameter to FindDefaultsEnd (the number of bytes to skip).

Example

```
var
  C : Cloner;
const
  Id : string[9] = 'MYPROG ID';
begin
  if not C.InitAndFind('MYPROG.EXE', UpdateAll, Id, SizeOf(Id)) then
    {handle error}
  else begin
    {... modify configuration data ...}
    C.Done;
  end;
end;
```

See Also

FindDefaultsEnd Init

Declaration

```
procedure LoadDefaults(FileOfs : LongInt; var Defaults; Bytes : Word);
```

Purpose

Seek to position FileOfs and read Defaults.

Description

This method is used to load configuration data after it has been located using FindDefaultsEnd, FindDefaultsStart, or InitAndFind. If an error occurs while loading the data, the error code may be obtained by calling GetLastError.

FileOfs, the starting offset in the file for the data to be read, will almost always be a value obtained by calling GetPos. Defaults indicates the start of the region into which the data is to be read, and Bytes is the number of bytes to read.

Important: GetPos returns the offset for the start of the ID string itself, not the offset for the "real" configuration data that follows it. Therefore, you should either 1) pass the name of the ID string as the Defaults parameter and pass the value returned by GetPos as the FileOfs parameter, or 2) pass the name of the first variable following the ID string as the Defaults parameter and pass GetPos+SizeOf(IdString) as the FileOfs parameter. If you choose 1), the Bytes parameter should include the size of the ID string; if you choose 2), it should not.

Note that LoadDefaults should *not* be called in a program that is modifying itself (as SMACS does), only in a program (such as DPINST) that is modifying another program (DESKPOP, in the case of DPINST).

Example

```
var
  C : Cloner;
const
  Id : string[9] = 'MYPROG ID';
  {... configuration data ...}
  CfgEnd : Byte = 0;
var
  SeekOfst : LongInt;
begin
  if not C.InitAndFind('MYPROG.EXE', UpdateAll, Id, SizeOf(Id)) then
    {handle error}
  else begin
    SeekOfst := C.GetPos;
    C.LoadDefaults(SeekOfst, Id, Ofs(CfgEnd)-Ofs(Id));
    {... modify configuration data and store it ...}
    C.Done;
  end;
end;
```

See Also

FindDefaultsEnd
FindDefaultsStart
GetPos
InitAndFind
StoreDefaults

StoreDefaults

Declaration

```
procedure StoreDefaults(FileOfs : LongInt; var Defaults; Bytes : Word);
```

Purpose

Seek to position FileOfs and store Defaults.

Description

This method is used to store updated configuration data in the EXE file. If an error occurs while storing the data, the error code may be obtained by calling GetLastError.

FileOfs, the starting offset in the file for the data to be written, will almost always be a value obtained previously by calling GetPos. Defaults indicates the start of the data to be written, and Bytes is the number of bytes to write.

As explained in detail in the entries for GetPos and LoadDefaults, GetPos returns the offset for the start of the ID string that was found, not the actual configuration data that follows it. Since this behavior has implications for StoreDefaults, you should be sure to read the documentation for those routines.

Example

See the example at the beginning of this section.

See Also

GetPos LoadDefaults

D. OPERRHAN: Runtime Error Recovery

Turbo Pascal offers a significant measure of flexibility in handling runtime errors . The programmer can disable range checking, stack overflow checking, and I/O result checking. Heap allocation errors can be diverted by using the HeapError facility. All runtime errors can be trapped to a user-defined exit procedure.

But one question remains unanswered: How can a program recover from a runtime error and continue processing? **Object Professional** provides the answer with the OPERRHAN unit.

OPERRHAN works by tapping into the Turbo Pascal runtime error handler. Control is diverted from the normal SYSTEM unit to a routine defined by OPERRHAN. The OPERRHAN routine classifies errors as recoverable or fatal, computes where control should return upon recovery, optionally calls a user-defined function, and finally returns to the program or to the SYSTEM error handler as required.

Some runtime errors are hopelessly bad--OPERRHAN recognizes these and passes control back to the SYSTEM error handler, which shuts down the program in an orderly fashion after calling any user-defined exit procedures. At present, we classify four types of Turbo runtime errors as fatal: error 202 (Stack overflow), error 203 (Heap overflow), errors 208 and 209 (overlay errors) and any error when the 80x87 numeric coprocessor (or Turbo Pascal's floating point emulator) is involved. All other Turbo Pascal runtime errors, including I/O errors, are considered recoverable.

Stack overflow problems can be solved by increasing the program's stack segment size via the {$M} compiler directive or by reorganizing the program's use of local variables. You can manage heap overflow errors in a recoverable way using the Turbo Pascal HeapError facility.

You can manage numeric coprocessor errors in a recoverable way by programming the 80x87's status register to return error codes rather than generating exception interrupts. See the documentation for the OP8087 unit for details.

Note that I/O errors (that is, runtime errors whose number is less than 200) will leave IoResult set to a non-zero value. IoResult must be cleared (by reading it) before additional I/O operations can be successfully executed. Because you can turn off Turbo's I/O checking with the {$I-} compiler directive, it is unlikely that you will need to use OPERRHAN for handling I/O errors. In some situations, however, the option of a centralized error handler may be desirable.

The following program demonstrates use of the OPERRHAN unit.

```
{$R+}
program TestErrHan;  {EXERRHAN.PAS}
uses
  Dos, OpErrHan, OpString;
var
  I, J, K : Integer;
  V : 1..10;

  {$F+} {Must use the FAR model}
  function RecoverError : Boolean;
    {-Invoked when an error occurs. Return True to enable recovery}
  var
    I : Word;
  begin
    {Clear IoResult}
    I := IoResult;
```

```
    WriteLn('Error handler sees error ', ExitCode:3, ' at ',
            HexW(Seg(ErrorAddr^)), ':', HexW(Ofs(ErrorAddr^)));
    RecoverError := True;
  end;

begin
  {Enable error recovery, and install user routine}
  ErrorRecovery(True, @RecoverError);

  I := 0;
  J := 1;
  K := J div I;
  if ExitCode <> 0 then
    WriteLn('Divide by zero error occurred');

  I := 20;
  V := I;
  if ExitCode <> 0 then
    WriteLn('Range error occurred');
end.
```

Variables

```
CanRecover : Boolean;
```

Because OPERRHAN must patch into Turbo's runtime system, its initialization code checks carefully to find the right patch point. If all goes well, OPERRHAN sets CanRecover to True. Presently we know of only two reasons why CanRecover might be False: either the Turbo SYSTEM library has changed radically as regards error handling, or the application program has taken control of INT $0 (divide by zero) or INT $23 (control break) prior to the time when OPERRHAN's initialization code is executed. The first problem necessitates changes to the OPERRHAN initialization code. The second can be solved by reordering the application's USES statement to place OPERRHAN before any user-defined units.

Declaration

```
procedure ErrorRecovery(On : Boolean; UserRoutine : Pointer);
```

Purpose

Turn error recovery on and off, and attach to a user error routine.

Description

Call this procedure to activate or deactivate runtime error recovery. If the On parameter is True, error recovery is activated; otherwise it is deactivated and Turbo Pascal's normal error handling system is used. OPERRHAN can recover from most but not all runtime errors. See the introduction to OPERRHAN for more details.

UserRoutine is a pointer to a user-defined function which can further customize the error handling behavior of an application. If you don't want to define such a user function, simply pass the value nil in this position. UserRoutine must point to a function declared as follows:

```
{$F+}
function RecoverError : Boolean;
  {-Invoked when an error occurs. Return True to enable recovery}
```

Note that the routine must be a global procedure compiled under the FAR model. Not preceding the function with the {$F+} directive will lead to a program crash.

The UserRoutine function will be called whenever a recoverable runtime error occurs. At that time the ExitCode global byte will contain the runtime error number, and the ErrorAddr global pointer will contain the normalized address where the runtime error occurred. UserRoutine can evaluate these variables and take any action required. Before exiting, UserRoutine should set its result True if recovery is desired, and False if the program should be terminated.

If a runtime error is fundamentally not recoverable, or if the UserRoutine returns False, then OPERRHAN passes control to the normal system error handler. This handler will operate as usual by calling any user-defined exit procedures, closing all files, restoring interrupt vectors, and halting.

If a runtime error is recoverable and the UserRoutine returns True, then OPERRHAN's routines will return control to the machine instruction immediately following the one that caused the error. The application program is responsible for regularly polling the ExitCode byte to determine whether an error has occurred. ExitCode will contain the normal values associated with Turbo runtime errors: zero for success, non-zero for failure. See the Turbo Pascal manual for a detailed listing of the error codes.

Whenever an application uses **Object Professional's** error recovery system, OPERRHAN's initialization code automatically calls ErrorRecovery to activate error recovery. By default, OPERRHAN installs no UserRoutine.

Example

```
{$F+}
function RecoverError : Boolean;
var I : Word;
begin
  I := IoResult;           {Clear IoResult}
  RecoverError := True;    {Recover from all errors}
end;
{$F-}
begin
  ErrorRecovery(True, @RecoverError);
  ...
```

This is the simplest useful UserRoutine. The routine clears IoResult to allow further I/O operations to proceed, and returns a True result to recover from all forms of runtime errors.

See Also

Error8087 (OP8087)

Exceptions8087 (OP8087)

E. OPINLINE: Assorted Inline Macros

One of the most powerful language extensions in Turbo Pascal is the "inline procedure." Unlike normal procedures and functions, whose code resides at a fixed location in memory, the entire contents of an inline procedure are inserted directly into the object code produced by the compiler each time it is referenced. Although inline procedures can take parameters and return function results, just like regular procedures and functions, the compiler treats them more like symbolic macros, such as those available in C. For this reason, we prefer to call them inline macros.

Among other things in OPINLINE, you will find inline macros that allow you to:

- Set a "jump marker" and later return to the location where the marker was set, from virtually any place in your program
- Set, clear, and test bit flags
- Normalize pointers
- Exchange the values in two variables without using a temporary variable
- Initialize an array of any type without using a FOR loop

You'll also find a variety of routines (most of which are not implemented as macros) for performing pointer arithmetic.

Types

```
JumpRecord  =
  record
    SpReg, BpReg : Word;
    JmpPt : Pointer;
  end;
```

Used by SetJump and LongJump to store data used to perform non-local goto's.

```
LH =
  record
    L, H : Word;
  end;
```

Used for typecasting long integers; allows easy access to both the high (H) and low (L) words that make up the LongInt.

```
OS =
  record
    O, S : Word;
  end;
```

Used for typecasting pointers; allows easy access to both the segment (S) and offset (O) portions of the address.

Declaration

```
function AddLongToPtr(P : Pointer; L : LongInt) : Pointer;
```

Purpose

Add a LongInt to a pointer, returning a normalized pointer.

Description

This routine converts the segmented address P to a linear address, adds L to it, then returns a segmented address that has been normalized.

Example

```
type
  BytePtr = ^Byte;
var
  P : Pointer; {points to a contiguous 0-based array of bytes 128K in size}
  L : LongInt;
begin
  {... allocate and initialize P ...}
  for L := 0 to $1FFFF do
    Write( BytePtr( AddLongToPtr(P, L) )^:4 );
end;
```

Writes the value of every Byte in the 128K array pointed to by P.

See Also

Normalized

Declaration

```
function AddWordToPtr(P : Pointer; W : Word) : Pointer;
```

Purpose

Add a Word to a pointer.

Description

AddWordToPtr is similar to AddLongToPtr, but no normalization or wrap checking is performed. In cases where either routine will do, AddWordToPtr will usually be preferred, because it is faster. (AddWordToPtr is implemented as an inline macro; AddLongToPtr is not.)

Example

```
P := Ptr($40, $01);
P1 := AddWordToPtr(P, $400);
P2 := AddLongToPtr(P, $400);
```

P1 would be set to $40:$401, and P2 would be set to $80:$01, because AddLongToPtr normalizes the pointers it returns and AddWordToPtr doesn't. The physical memory location is the same in both cases.

See Also

Normalized

Array2Str / ByteFlagIsSet

Declaration

```
function Array2Str(var A; Len : Byte) : string;
```

Purpose

Convert an array of char to a string.

Description

Quickly converts an array of characters (widely used in database applications) to a string. Because this is an inline macro, it is slightly faster than Turbo Pascal's built-in conversion routine, which requires a call to the SYSTEM unit.

Examples

```
const
  A : array[1..4] of Char = 'TEST';
var
  S : string;
...
  S := Array2Str(A, 4);
```

S would be set to 'TEST'. S[0] would contain #4.

```
FastWrite(Array2Str(A, 4), 1, 1, $07);
```

Displays 'TEST' at top left corner of the screen.

Declaration

```
function ByteFlagIsSet(Flags, FlagMask : Byte) : Boolean;
```

Purpose

Returns True if the bit specified by FlagMask is set in Flags.

Description

This macro tests to see if any of the bits set in FlagMask are also set in Flags. In most cases, FlagMask should not have more than one bit set. Although ByteFlagIsSet can tell you if one of them is set, or none of them, it cannot tell you if all of them are set.

Examples

```
ByteFlagIsSet(BitFlags,$02);
```

Correct. This would return True if bit 1 in BitFlags is set ($02 = 00000010 in binary).

```
ByteFlagIsSet(BitFlags,$03);
```

Probably incorrect. This will not tell you if bits 0 and 1 are both set ($03 = 00000011 in binary). It will only tell you if one of them is set.

See Also

ClearByteFlag SetByteFlag

Declaration

```
procedure CallOldIsr(OldIsr : Pointer);
```

Purpose

Call previous ISR from an interrupt procedure.

Description

Simulates an interrupt by pushing the CPU flags and calling FAR to the address passed as OldIsr. Like EmulateInt (OPINT), this macro thus allows you to call the ISR that serviced a particular interrupt vector before you grabbed it. Unlike EmulateInt, however, it is intended primarily for use in ISR's written in inline code, rather than Pascal. Note that the BX register is destroyed by this macro.

Example

```
procedure MyIsr(BP : Word); interrupt;
begin
  CallOldIsr(SaveInt09);
  ...
end;
```

Calls the previous interrupt $09 vector (address in SaveInt09).

See Also

EmulateInt (OPINT) JumpToOldIsr

Declaration

```
procedure ClearByteFlag(var Flags : Byte; FlagMask : Byte);
```

Purpose

Clear the bit(s) specified by FlagMask in Flags.

Description

This macro clears all bits in Flags that are set in FlagMask. Note that the Flags parameter must be a variable of type Byte.

Example

```
ClearFlag(BitFlags, $02);
Clears bit 1 in BitFlags ($02 = 00000010 in binary).
```

See Also

ByteFlagIsSet SetByteFlag

Declaration

```
procedure ClearFlag(var Flags : Word; FlagMask : Word);
```

Purpose

Clear the bit(s) specified by FlagMask in Flags.

Description

This macro clears all bits in Flags that are set in FlagMask. Note that the Flags parameter must be a variable of type Word.

Example

```
ClearFlag(BitFlags, $0002);
```

Clears bit 1 in BitFlags ($0002 = 0000000000000010 in binary).

See Also

FlagIsSet SetFlag

ClearLongFlag / ExchangeBytes / ExchangeLongInts

Declaration

```
procedure ClearLongFlag(var Flags : LongInt; FlagMask : LongInt);
```

Purpose

Clear the bit(s) specified by FlagMask in Flags.

Description

This macro clears all bits in Flags that are set in FlagMask. Note that the Flags parameter must be a variable of type LongInt.

Example

```
ClearLongFlag(BitFlags, $00000002);
```

Clears bit 1 in BitFlags ($00000002 = 00000000000000000000000000000010 in binary).

See Also

LongFlagIsSet SetLongFlag

Declaration

```
procedure ExchangeBytes(var I, J : Byte);
```

Purpose

Exchange bytes I and J.

Description

This macro allows you to exchange the values in I and J without using a temporary variable. It is especially useful in sorts. Note, however, that using a temporary variable will be faster than using ExchangeBytes unless the variables being exchanged are elements in an array.

Example

```
if MyArray[I] > MyArray[I+1] then
  ExchangeBytes(MyArray[I], MyArray[I+1]);
```

Exchange MyArray[I] and MyArray[I+1] if MyArray[I] is greater.

Declaration

```
procedure ExchangeLongInts(var I, J : LongInt);
```

Purpose

Exchange longints I and J.

Description

This macro allows you to exchange the values in I and J without using a temporary variable. It is especially useful in sorts. Note, however, that using a temporary variable will be faster than using ExchangeLongInts unless the variables being exchanged are elements in an array.

Note that ExchangeLongInts may also be used with pointer variables by typecasting each pointer to a LongInt, as shown in the second example below.

Examples

```
if MyArray[I] > MyArray[I+1] then
  ExchangeLongInts(MyArray[I], MyArray[I+1]);
```

Exchange MyArray[I] and MyArray[I+1] if MyArray[I] is greater.

```
if MyPtrArray[I]^ > MyPtrArray[I+1]^ then
  ExchangeLongInts( LongInt(MyPtrArray[I]), LongInt(MyPtrArray[I+1]) );
```

Exchange MyPtrArray[I] and MyPtrArray[I+1] if MyPtrArray[I]^ is greater.

See Also

ExchangeWords ExchangeStructs

Declaration

```
procedure ExchangeStructs(var I, J; Size : Word);
```

Purpose

Exchange structures I and J.

Description

This macro allows you to exchange the values in I and J without using a temporary variable. I and J may be variables of any type, but they must be the same Size. It is especially useful in sorts. Note, however, that using a temporary variable will generally be faster than using ExchangeStructs. ExchangeStructs is best used in cases where you cannot afford to set aside space for a temporary variable.

Examples

```
if MyRec1.Field > MyRec2.Field then
  ExchangeStructs(MyRec1, MyRec2, SizeOf(MyRec1));
```

Exchange MyRec1 and MyRec2 if MyRec1.Field is greater than MyRec2.Field.

```
if MyString1 > MyString2 then begin
  I := MaxWord(Length(MyString1), Length(MyString2));
  ExchangeStructs(MyString1, MyString2, I+1);
end;
```

Exchange MyString1 and MyString2 if MyString1 is greater. Notice that the number of bytes moved is limited to the length of the longer string plus 1 (to account for the length bytes of the two strings).

See Also

ExchangeBytes ExchangeLongInts

Declaration

```
procedure ExchangeWords(var I, J : Word);
```

Purpose

Exchange words I and J.

Description

This macro allows you to exchange the values in I and J without using a temporary variable. It is especially useful in sorts. Note, however, that using a temporary variable will be faster than using ExchangeWords unless the variables being exchanged are elements in an array.

Example

```
if MyArray[I] > MyArray[I+1] then
  ExchangeWords(MyArray[I], MyArray[I+1]);
```

Exchange MyArray[I] and MyArray[I+1] if MyArray[I] is greater.

See Also

ExchangeBytes ExchangeLongInts

FarCall / FillStruct

Declaration

```
procedure FarCall(ProcAddr : Pointer);
```

Purpose

ProcAddr is the address of a routine to be called FAR.

Description

This macro lets you call any procedure in any code segment as long as 1) it takes no parameters, 2) it was compiled under the FAR model using {$F+}, and 3) it is not nested inside another procedure. FarCall might be used, for example, to implement a jump table. This routine is provided only for backward compatibility with Turbo Professional.

Examples

```
FarCall(ProcPtr);
```

Correct. This would make a FAR call to MyProc.

```
ProcPtr := @MyProc;
...
FarCall(ProcPtr^);
```

Incorrect! The '^' is not needed.

```
var MyJumpTable : array[1..10] of Pointer;
...
FarCall(MyJumpTable[Index]);
```

Make a FAR call to the procedure whose address is in MyJumpTable[Index].

See Also

NearCall

Declaration

```
procedure FillStruct(var Dest; Count : Word; var Filler; FillerSize : Word);
```

Purpose

Fill memory starting at Dest with Count instances of Filler.

Description

This macro is similar to the standard FillChar procedure, except that it fills memory with a value of FillerSize bytes rather than a byte-sized value. Dest is the location in memory to start filling at. Filler is the value to fill with. Its size is given by FillerSize. Count is the number of instances of Filler. FillStruct is especially useful for initializing arrays. Note that no range checking is done to prevent segment overruns.

Example

```
var   MyArray : array[1..1000] of Real;
const R : Real = 1.2345;
...
FillStruct(MyArray, 1000, R, SizeOf(R));
```

Initialize all 1000 elements of MyArray to 1.2345.

Declaration

```
procedure FillWord(var Dest; Count, Filler : Word);
```

Purpose

Fill memory starting at Dest with Count instances of Filler.

Description

This macro is similar to the standard FillChar procedure, except that it fills memory with a word-sized value rather than a byte-sized value. Dest is the location in memory to start filling at. Filler is the word value to fill with. And Count is the number of words to fill. FillWord is especially useful for initializing arrays. Note that no range checking is done to prevent segment overruns.

Example

```
var
  MyArray1 : array[1..100] of Word;
  MyArray2 : array[1..100] of Integer;
  I : Integer;
...
FillWord(MyArray1, 100, 1);
I := 1;
FillWord(MyArray2, 100, Word(I));
```

Initialize all 100 elements of MyArray1 and MyArray2 to 1.

Declaration

```
function FlagIsSet(Flags, FlagMask : Word) : Boolean;
```

Purpose

Returns True if the bit specified by FlagMask is set in Flags.

Description

This macro tests to see if any of the bits set in FlagMask are also set in Flags. In most cases, FlagMask should not have more than one bit set. Although FlagIsSet can tell you if one of them is set, or none of them, it cannot tell you if all of them are set.

Examples

```
FlagIsSet(BitFlags,$0002);
```

Correct. This would return True if bit 1 in BitFlags is set ($0002 = 0000000000000010 in binary).

```
FlagIsSet(BitFlags,$0003);
```

Probably incorrect. This will not tell you if bits 0 and 1 are both set ($0003 = 0000000000000011 in binary). It will only tell you if one of them is set.

See Also

ClearFlag SetFlag

HiWord / JumpToOldIsr

Declaration

```
function HiWord(L : LongInt) : Word;
```

Purpose

Return high-order word of L.

Description

This macro returns the high-order word of a long integer. HiWord is found in the Macintosh version of Turbo Pascal, but not in the IBM version.

Example

```
const MyLongInt = $FFFF0000;
...
MyWord := HiWord(MyLongInt);
```

This would assign $FFFF to MyWord.

See Also

LoWord SwapWord

Declaration

```
procedure JumpToOldIsr(OldIsr : Pointer);
```

Purpose

Jump to previous ISR from an interrupt procedure.

Description

Like ChainInt (OPINT), this macro allows you to restore the CPU registers pushed onto the stack at the beginning of an interrupt procedure, then pass on the interrupt to the ISR that serviced a particular vector before you grabbed it. Unlike ChainInt, however, it is intended primarily for use in ISR's written in inline code, rather than Pascal.

Examples

```
GetIntVec($09, SaveInt09);
...
procedure MyIsr(BP : Word); interrupt;
begin
  {-- something --}
  JumpToOldIsr(SaveInt09);
end;
```

Correct. Jumps to the previous interrupt $09 handler.

```
JumpToOldIsr(SaveInt09^);
```

Incorrect! The '^' is not needed.

See Also

CallOldIsr ChainInt (OPINT)

Declaration

```
function LongFlagIsSet(Flags, FlagMask : LongInt) : Boolean;
```

Purpose

Returns True if the bit specified by FlagMask is set in Flags.

Description

This macro tests to see if any of the bits set in FlagMask are also set in Flags. In most cases, FlagMask should not have more than one bit set. Although LongFlagIsSet can tell you if one of them is set, or none of them, it cannot tell you if all of them are set.

Examples

```
LongFlagIsSet(BitFlags,$00000002);
```

Correct. This would return True if bit 1 in BitFlags is set ($00000002 = 00000000000000000000000000000010 in binary).

```
LongFlagIsSet(BitFlags,$00000003);
```

Probably incorrect. This will not tell you if bits 0 and 1 are both set ($00000003 = 00000000000000000000000000000011 in binary). It will only tell you if one of them is set.

See Also

ClearLongFlag SetLongFlag

Declaration

```
procedure LongJump(var JumpDest : JumpRecord; Result : Word);
```

Purpose

Restore SP, BP, and jump to JumpDest.JmpPt.

Description

This macro uses the data stored in the JumpRecord by SetJump to perform a non-local goto. It restores the stack frame to the condition it was in when SetJump was called, then jumps to the instruction immediately after the call. Result is the value to be returned as SetJump's "function result" (see the entry for SetJump).

Very important: LongJump cannot be used to jump from a higher stack frame into a lower one. Consider the following case, for example:

```
program MyProg;
  procedure MyProc;
  begin
    {Point B}
  end;
begin
  {Point A}
end.
```

It would be perfectly OK to call SetJump at Point A and later call LongJump from Point B to return to Point A. You could not, however, do the reverse (jump to B from A). Although LongJump uses a FAR jump instruction, the issue of FAR model {$F+} vs. NEAR model {$F-} does not arise here, as it does with call instructions.

Example

See the example for SetJump.

See Also

SetJump

LongToPtr / LoWord / MakeInteger

Declaration
```
function LongToPtr(L : LongInt) : Pointer;
```

Purpose
Return LongInt L as a normalized pointer.

Description
Converts L, which is assumed to represent a linear address, to a normalized pointer (a segmented address).

Examples
```
LongToPtr($401)
```
Returns $40:$01.
```
P := LongToPtr( PtrToLong(Ptr($40, $01)) + $100 );
```
P is set to $50:$01;

See Also
PtrToLong

Declaration
```
function LoWord(L : LongInt) : Word;
```

Purpose
Return low-order word of L.

Description
This macro returns the low-order word of a long integer. LoWord is found in the Macintosh version of Turbo Pascal, but not in the IBM version.

Example
```
const MyLongInt = $FFFF0000;
...
MyWord := LoWord(MyLongInt);
```
This would assign $0000 to MyWord.

See Also
HiWord SwapWord

Declaration
```
function MakeInteger(H, L : Byte): Integer;
```

Purpose
Constructs an integer from two bytes.

Description
Places H in the high byte of the function result, L in the low byte. Implementation is identical to that of MakeWord.

Examples
```
MakeInteger(1, 2)
```
Returns 258 ($0102).
```
MakeInteger(255, 0)
```
Returns -256 ($FF00).

See Also
MakeLongInt MakeWord

Declaration
```
function MakeLongInt(H, L : Word) : LongInt;
```

Purpose
Constructs a LongInt from two Words.

Description
Places H in the high word of the function result, L in the low word.

Examples
```
MakeLongInt(1, 2)
```
Returns 65538 ($00010002).
```
MakeLongInt(65535, 0)
```
Returns -65536 ($FFFF0000).

See Also
MakeInteger

Declaration
```
function MakeWord(H, L : Byte) : Word;
```

Purpose
Constructs a word from two bytes.

Description
Places H in the high byte of the function result, L in the low byte. Implementation is identical to that of MakeInteger.

Examples
```
MakeWord(1, 2)
```
Returns 258 ($0102).
```
MakeWord(255, 0)
```
Returns 65280 ($FF00).

See Also
MakeInteger

Declaration
```
function MaxLong(A, B : LongInt) : LongInt;
```

Purpose
Returns the greater of A and B.

Description
This macro does a signed comparison of A and B and returns the greater of the two numbers.

Examples
```
MaxLong(MaxLongInt, 0)
```
Returns MaxLongInt.
```
MaxLong(-1, -1000)
```
Returns -1.

See Also
MinLong MaxWord

MaxWord / MinLong / MinWord

Declaration

```
function MaxWord(A, B : Word) : Word;
```

Purpose

Returns the greater of A and B.

Description

This macro does an unsigned comparison of A and B and returns the greater of the two numbers.

Example

```
I := MaxWord(12, 40000);
```

I would be set to 40000.

See Also

MaxLong MinWord

Declaration

```
function MinLong(A, B : LongInt) : LongInt;
```

Purpose

Returns the smaller of A and B.

Description

This macro does a signed comparison of A and B and returns the lesser of the two numbers.

Example

```
L := MinLong(12, -400000);
```

L would be set to -400000.

See Also

MaxLong

Declaration

```
function MinWord(A, B : Word) : Word;
```

Purpose

Returns the smaller of A and B.

Description

This macro does an unsigned comparison of A and B and returns the lesser of the two numbers.

Example

```
I := MinWord(12, 40000);
```

I would be set to 12.

Declaration
```
procedure NearCall(ProcOfs : Word);
```

Purpose
ProcOfs is the offset of a routine to be called near.

Description
This macro lets you call any procedure in the current code segment as long as 1) it takes no parameters, 2) it was compiled using the NEAR model (selected with the {$F-} compiler directive, the default setting), and 3) it is not nested inside another procedure. It might be used, for example, to implement a jump table.

Example
```
var MyJumpTable : array[1..10] of Word;
...
NearCall(MyJumpTable[I]);
```
Makes a NEAR call to the procedure whose offset is stored in MyJumpTable[I].

See Also
FarCall

Declaration
```
function Normalized(P : Pointer) : Pointer;
```

Purpose
Return P as a normalized pointer.

Description
This macro "normalizes" a pointer value. That is, it adjusts the segment and offset such that the offset is in the range $0000-$000F.

Examples
```
P := Normalized( Ptr($0000, $0410) );
```
This would point P to $0041:$0000.
```
P := Normalized( Ptr($0040, $0113) );
```
This would point P to $0051:$0003.

Declaration
```
function PtrDiff(P1, P2 : Pointer) : LongInt;
```

Purpose
Return the number of bytes between P1^ and P2^.

Description
This routine converts the two segmented addresses passed as parameters into linear addresses (using PtrToLong) and then returns the absolute difference between them.

Examples
```
P1 := Ptr($40, $10); P2 := Ptr($41, $01);
L := PtrDiff(P1, P2);
```
L would be set to 1 ($40:$10 = $41:$00).
```
L := PtrDiff(P2, P1);
```
L would still be set to 1; the order of the parameters doesn't matter.

See Also
PtrToLong

PtrToLong / Reboot / SetByteFlag

Declaration

```
function PtrToLong(P : Pointer) : LongInt;
```

Purpose

Convert pointer, in range $0:$0 to $FFFF:$000F, to a LongInt.

Description

This routine converts segmented addresses (pointers) into linear addresses (longints). It is most typically used in conjunction with LongToPtr, when performing pointer arithmetic. It is also used by PtrDiff when calculating the number of bytes between two segmented addresses.

A linear address is calculated by multiplying the segment portion of a pointer by 16 ($10) and adding the offset portion to the result.

Example

```
P := Ptr($40, $10);
L := PtrToLong(P);
```

L would be set to $410.

See Also

LongToPtr PtrDiff

Declaration

```
procedure Reboot;
```

Purpose

Reboot the machine.

Description

This macro forces the machine to reboot itself. It works by storing the "magic value" $1234 at. address $40:$72 and then jumping to $FFFF:$0000. On all IBM PC's and compatibles, the latter address contains a jump instruction that leads to the system's hardware test routines and the ROM BIOS's bootstrap code.

Declaration

```
procedure SetByteFlag(var Flags : Byte; FlagMask : Byte);
```

Purpose

Set the bit(s) specified by FlagMask in Flags.

Description

This macro sets all bits in Flags that are set in FlagMask. Note that the Flags parameter must be a Byte variable.

Examples

```
SetByteFlag(BitFlags, $02);
```

Sets bit 1 in BitFlags ($02 = 00000010 in binary).

```
SetByteFlag(BitFlags, $07);
```

Sets bits 2, 1, and 0 in BitFlags ($07 = 00000111 in binary).

See Also

ByteFlagIsSet ClearByteFlag

Declaration

```
procedure SetFlag(var Flags : Word; FlagMask : Word);
```

Purpose

Set the bit(s) specified by FlagMask in Flags.

Description

This macro sets all bits in Flags that are set in FlagMask. Note that the Flags parameter must be a Word variable.

Examples

```
SetFlag(BitFlags, $0002);
```

Sets bit 1 in BitFlags ($0002 = 0000000000000010 in binary).

```
SetFlag(BitFlags, $0007);
```

Sets bits 2, 1, and 0 in BitFlags ($0007 = 0000000000000111 in binary).

See Also

ClearFlag FlagIsSet

Declaration

```
function SetJump(var JumpDest : JumpRecord) : Word;
```

Purpose

Save current SP, BP, and a jump destination.

Description

This macro saves data about the current stack frame and the instruction pointer, data which can later be used by LongJump to restart execution at the instruction immediately following the call to SetJump. A typical use for it might be to make a call to SetJump just before the start of the main program loop. Any time thereafter that an error occurred, you could jump back to the start of the loop using LongJump.

When SetJump is actually called, it returns 0 to indicate that the jump record was just initialized. If the value returned is anything other than 0, the value was passed as the Result parameter to LongJump. Although you are free to ignore the function result if you wish, the ability to discriminate between the initial call to SetJump and subsequent "calls" made by LongJump is quite useful, because it allows LongJump to pass error codes and other information to the routine containing the call to SetJump.

For good examples of how to use SetJump and LongJump, see the source code for the OPBROWSE unit.

Example

```
var
  MyJumpRecord : JumpRecord;
  I : Word;
...
  I := SetJump(MyJumpRecord);
  if I <> 0 then
    WriteLn('This is JumpDest.JmpPt. Result = ', I);
  {-- something --}
  if I < 10 then
    LongJump(MyJumpRecord, I+1);
```

The call to LongJump would restart execution at the location of the "if I < > 0" test. I would be set to 0 when SetJump was first called, and to increasingly higher values after calls to LongJump. The sequence would end after 10 iterations.

See Also

LongJump

SetLongFlag / SwapNibble / SwapWord

Declaration

```
procedure SetLongFlag(var Flags : LongInt; FlagMask : LongInt);
```

Purpose

Set the bit(s) specified by FlagMask in Flags.

Description

This macro sets all bits in Flags that are set in FlagMask. Note that the Flags parameter must be a LongInt variable.

Examples

```
SetLongFlag(BitFlags, $00000002);
```

Sets bit 1 in BitFlags ($00000002 = 00000000000000000000000000000010 in binary).

```
SetLongFlag(BitFlags, $00000007);
```

Sets bits 2, 1, and 0 in BitFlags ($00000007 = 00000000000000000000000000000111 in binary).

See Also

ClearLongFlag LongFlagIsSet

Declaration

```
function SwapNibble(B : Byte) : Byte;
```

Purpose

Swap the high and low nibbles of B.

Description

This routine is similar to the built-in Swap routine, but instead of swapping the high and low bytes of a word-sized value, it swaps the high and low nibbles of a byte-sized value. (A "nibble" is four bits, or half a byte.)

Examples

```
SwapNibble($F0)
```

Returns $0F (00001111 in binary; $F0 = 11110000 in binary).

```
SwapNibble($81)
```

Returns $18 (00011000 in binary; $81 = 10000001 in binary).

Declaration

```
function SwapWord(L : LongInt) : LongInt;
```

Purpose

Swap the high and low words of L.

Description

This routine is similar to the built-in Swap routine, but instead of swapping the high and low bytes of a word-sized value, it swaps the high and low words of a LongInt-sized value.

Example

```
SwapWord($FFFF0000)
```

Returns $0000FFFF.

```
SwapWord($12345678)
```

Returns $56781234.

F. OPSORT: In-memory Sort Routine

Sorting is a common task performed by many programs. Uses for sorting range from listing addresses alphabetically to computing statistics on sets of numbers.

Object Professional provides a sort routine that is fast and easy to use. With it, you can sort up to 65,535 data elements of any type. All elements must fit into normal memory, however, since paging to disk is not supported. The OPSORT unit is quite compact, using less than 2000 bytes of code space. It implements a non-recursive quicksort that also minimizes use of stack space.

If you have seen Borland's Database Toolbox sort system, using the OPSORT unit will be a snap since the two are organized in a similar fashion. OPSORT differs primarily in implementation. As mentioned above, OPSORT does not support paging to disk--instead it provides a faster, more compact sort of adequate capacity for almost every situation. OPSORT has a special mode for sorting strings--instead of allocating space for the maximum length of a string, it allocates only the actual length. By using procedure pointers, OPSORT also makes the sort completely dynamic, that is, one set of sort routines can sort many data types within the same program.

OPSORT exports three routines: Sort, PutElement, and GetElement. Your program will call Sort first. Sort requires several parameters, as described in detail in the following sections. Two of the parameters tell Sort about the elements to be sorted--their size and the maximum number to expect. Three parameters provide the names of routines that Sort will call during the course of execution--one routine that will load elements into Sort's data structures, another that Sort will call to compare pairs of elements, and a final one to return the sorted results to your program.

The routines you provide to load and retrieve elements will call the other two functions exported by OPSORT. For each element to be sorted, your load routine will call PutElement to enter the data. For each sorted element retrieved, your routine will call GetElement.

The following example demonstrates the process of sorting a group of integers:

```
program SortInt;  {EXSORT.PAS}
  {-Sort a random set of integers}
uses
  OpSort;
const
  NumToSort = 1000;
var
  Status : SortStatus;

{$F+}
procedure GetInts;
  {-Load the sort data structures with elements to be sorted}
var
  N : Word;
  Value : Integer;
begin
  for N := 1 to NumToSort do begin
    Value := Random(MaxInt);
    if not PutElement(Value) then
      {Sort error occurred - most likely out of memory}
      Exit;
  end;
end;
```

```
function Less(var X, Y) : Boolean;
  {-Compare two integers}
var
  XI : Integer absolute X;
  YI : Integer absolute Y;
begin
  Less := (XI < YI);
end;

procedure WriteInts;
  {-Write the sorted output}
var
  Value : Integer;
begin
  while GetElement(Value) do
    WriteLn(Value);
end;
{$F-}

begin
  Status := Sort(NumToSort, SizeOf(Integer), GetInts, Less, WriteInts);
  case Status of
    SortSuccess : WriteLn(ElementsSorted, ' elements sorted');
    SortOutOfMemory : WriteLn('Insufficient memory');
    SortTooManyElements : WriteLn('Too many elements to sort');
  end;
end.
```

Refer to the sections following for further details.

OPSORT and Memory Management

Normally OPSORT uses a special kind of Mark/Release technique to allow rapid deallocation of heap space without destroying the Turbo memory manager's free list. This technique works fine so long as the caller of the OPSORT routines does not need to perform heap allocation or deallocation while the sort is in progress--that is, within the user-defined procedures of the sort. If you do need to do so, you will have to modify the source code for OPSORT. We have made this change a simple one, however: all you need to do is un-define the conditional compilation directive FastDispose in OPDEFINE.INC, then recompile OPSORT.

Types

```
GetPutProc = procedure;
```

A procedure used to load or retrieve the elements being sorted.

```
SortStatus  =
  (SortSuccess,              {Successful sort}
   SortOutOfMemory,          {Insufficient memory}
   SortTooManyElements);     {Attempt to sort more elements than requested}
```

The Sort function returns a result of the type SortStatus. The calling application should check this result to determine the outcome of the sort.

```
UserLessFunc = function (var X, Y) : Boolean;
```

A function to compare two elements.

Variables

```
ElementsSorted  : Word;
```

When a sort has been completed, the ElementsSorted variable will have been initialized to contain the actual number of elements that were sorted.

Declaration

```
function GetElement(var X) : Boolean;
```

Purpose

Get next element from the sorted array.

Description

Call GetElement to retrieve each element in sorted order after a sort has been completed. Your application will have the opportunity to call GetElement within the retrieval routine whose name you pass to the Sort function.

Because OPSORT does not know the type of your data, GetElement's var parameter is untyped. It is your responsibility to provide a variable of the proper type. OPSORT knows how many bytes to pass back based on a parameter passed to the Sort function.

GetElement returns True as long as there are further elements to retrieve. Once it returns False, the parameter X is undefined.

Example

```
while GetElement(S) do
  WriteLn(S);
```

This example simply writes sorted elements (of any simple type) to the screen.

See Also

Sort

PutElement

Declaration

```
function PutElement(var X) : Boolean;
```

Purpose

Put a sort element into the sort array.

Description

Call PutElement to add each new element to OPSORT's internal sort structure. Your application will have the opportunity to call PutElement within the loader routine whose name you pass to the Sort function.

Because OPSORT does not know the type of your data, the parameter to PutElement is an untyped var parameter. As such, you cannot pass a constant or expression to PutElement. However, the parameter can be almost any type of variable: a simple numeric element, a pointer to another structure, a variable length string, just about anything. OPSORT knows how many bytes to transfer based on the initial call to Sort.

PutElement returns True when an element is successfully added to the sort array. If it returns False, your application should stop adding elements and exit the loader routine. PutElement will return False if insufficient heap space is available for the new element, or if the number of elements exceeds the maximum allotted during the call to Sort.

Examples

```
while not Eof(F) do begin
  ReadLn(F, S);
  if not PutElement(S) then
    Exit;
end;
```

The example reads lines from a text file and adds each one to the sort array. If PutElement returns False, the loader routine is exited.

```
for I := 1 to NumPointers do
  if not PutElement(P[I]) then
    Exit;
```

In this example, P is an array of pointers to another data structure, perhaps a record. The pointers themselves are added to the sort rather than the full records (to save memory space). The application's element comparison function will dereference the pointers to perform the comparison.

See Also

Sort

Declaration

```
function Sort(Elements, ElementSize : Word; GetElements : GetPutProc;
              LessFunc : UserLessFunc; PutElements : GetPutProc) : SortStatus;
```

Purpose

Sort a group of elements.

Description

Sort is the main routine in OPSORT. An application will call Sort prior to calling any of the other routines.

Sort is designed to be completely configurable at runtime. Within memory limitations, it can sort up to 65,535 elements of any type, calling user-defined functions for element comparison, input and output. Sort keeps all elements in memory. This choice provides faster operation than a virtual sort and adequate capacity for most tasks.

The first two parameters to Sort tell it how many elements to expect, and the size of each element. The Elements parameter specifies the maximum number to expect. It is no problem if fewer elements are actually available. If more elements are encountered, a sort error occurs.

In all but one special case, ElementSize specifies the number of bytes in each sort element. This size is best obtained by using Turbo's built-in operator SizeOf. For example, SizeOf(Pointer) returns 4, and SizeOf(MyDataType) returns the number of bytes in a given data type. Sort provides a special case for strings. If you specify an ElementSize of 0, Sort interprets each element as a Turbo string and allocates memory for just the size required by each string.

Programs using OPSORT must be structured in a certain way, as exemplified by the simple example at the beginning of this chapter. The application must define three routines whose names are passed to Sort as procedure pointers.

```
{$F+}
procedure GetSortElements;
  {-Load elements from user data into the Sort routine}
begin
  ...
  if not PutElement(X) then
    Exit;
  ...
end;

procedure PutSortElements;
  {-Return sorted elements to the user program}
begin
  ...
  while GetElement(X) do
    {use the element X} ;
  ...
end;

function Less(var X, Y) : boolean;
begin
  {Return boolean for X < Y}
end;
{$F-}
```

Note that the three routines must be of the same type (function or procedure) as shown and take the same parameters as shown (the names of the routines are unimportant). They must be global procedures--that is, they cannot be nested inside other procedures--and they must be compiled under the FAR model. For this example, Sort would be called as follows:

```
Status := Sort(Elements, ElementSize, GetSortElements, Less, PutSortElements);
```

Sort

Sort will call your GetSortElements routine after performing some error checking and initialization tasks. GetSortElements in turn calls PutElement to put each element into OPSORT's internal data structure. When GetSortElements returns, Sort actually performs the sorting operation. Whenever Sort needs to compare two elements to place them into sorted order, it calls your Less function. The Less function compares the elements in any desired way depending on the data type, and returns True if the first element X is "less than" the second element Y. After sorting is complete, Sort calls your PutSortElements routine, which in turn calls GetElement to get each element back in sorted order. When PutSortElements returns, Sort deallocates the memory it has used and returns a sort status to the original calling program. See the SortStatus type for the codes Sort uses.

Examples

See the example program at the beginning of this chapter for a more extensive example.

15. Appendices

A. What's New in Object Professional 1.0

This section is intended to give owners of Turbo Professional an overview of some of the major differences between it and **Object Professional**. If you do not own Turbo Professional, you can skip this section of the manual. In writing it, we have addressed ourselves to the differences between **Object Professional** and Turbo Professional 5.0, but users of Turbo Professional 4.0 should also find this information helpful.

A complete list of all the features that are new in **Object Professional** would run to several pages, but here are some of the highlights:

- container classes: stacks, queues, linked lists, bit sets, string dictionaries
- object-oriented disk I/O
- random access object libraries
- large arrays that use RAM, disk, or EMS for storage, depending on what's available
- a swapping TSR manager to fit memory resident programs in only 6K of RAM
- a swapping Exec function
- low- and high-level printer control
- customizable printed reports
- windows that can be moved, zoomed, and resized
- greatly improved mouse support
- text and binary file browsing
- complete window-oriented text editing (word wrap, find/replace, block operations)
- pathname lists
- a MAKEMENU-like utility for designing data entry screens

That list covers most of the major enhancements, but there are many, many minor enhancements as well, particularly in the units that fall into the Screen Management (Volume 1) and Data Input/Output (Volume 2) categories.

Before we turn our attention to units that are new or different, here are the units that either haven't changed at all, or have received comparatively minor enhancements:

```
OP8087
OPASCIIZ
OPBCD
OPCRT
OPDATE
OPDOS
OPEMS
OPENHKBD
OPERRHAN
OPEXTMEM
OPINLINE
OPINT
OPMACED
OPMACRO
OPMOUSE
OPSORT
OPSTRDEV
OPSTRING
OPTSR
```

The names have been changed for the sake of consistency, but for the most part these units are either the same as their TPxxx counterparts or provide a superset of their functions. The one possible exception here is OPCRT, since all of the window-relative screen writing routines in TPCRT (FastWriteWindow, FastReadWindow, etc.) have been incorporated into OPWINDOW.

The following Turbo Professional units are gone, either because they were deemed unsuitable for **Object Professional** or because their functionality was incorporated into another unit (shown in parentheses):

```
TPCMDLIN
TPEARRAY (OPLARRAY)
TPINT24
TPMEMCHK (OPROOT)
TPRARRAY (OPLARRAY)
TPSCREEN
TPVARRAY (OPLARRAY)
```

Quite a few of the Turbo Professional demonstration programs are gone as well:

```
BSORT
DIFF
PCALC
PREF
PTRPT
REP
TFIND
VBROWSE
WINEXMPL
```

BSORT, DIFF, REP, and TFIND were dropped to make room for new demo programs that illustrate object oriented programming techniques and/or use one of the new units. (All four could be converted to **Object Professional** quite easily, but there's little reason to do so.) The functionality of PREF and PCALC has been incorporated into DESKPOP. PTIME has been changed to eliminate the need for PTRPT. WINEXMPL was dropped at the same time as TPSCREEN. And VBROWSE is effectively replaced by BROWSE. The other Turbo Professional demo programs--BEST, ENTRY, MAKEHELP, MAKEMENU, MEMO, PTIME, SMACS, and WINWOW--appear in **Object Professional** in revised form.

The following units are based on ones in Turbo Professional, but they have all been completely rewritten to use objects:

```
OPCMD
OPDIR
OPEDIT
OPENTRY
OPHELP
OPLARRAY
OPMEMO
OPMENU
OPPICK
OPWINDOW
```

OPLARRAY combines the functions of TPEARRAY, TPRARRAY, and TPVARRAY into one unit. OPCMD is based on TPCMD, an undocumented unit that was secretly used by several Turbo Professional units. OPMEMO is based on TPMEMO, which was introduced in Turbo Professional 5.02.

The following units are, for the most part, completely new:

```
OPABSFLD
OPBROWSE
OPCLONE
OPCOLOR
OPDEVICE
OPEDITOR
OPEXEC
OPFEDIT
OPFIELD
OPFORM
OPFRAME
OPPRINT
OPPRNLOW
OPREPLAY
OPROOT
OPSEDIT
OPSELECT
OPSWAP
OPSWAP1
```

OPABSFLD, OPFEDIT, OPFIELD, OPSELECT, and OPSEDIT all have some of the elements of TPENTRY and/or TPEDIT in them. OPFRAME is based in part on code that was in TPWINDOW. OPROOT contains some code that used to be in TPMEMCHK, but everything else in the unit is new. OPREPLAY is essentially a cut-down version of OPMACRO, which is largely identical to TPMACRO. OPCLONE is based on TPCLONE, a unit of ours that appeared on the Turbo Professional BONUS disk. OPCOLOR is comparable to COLORDEF, another unit (not written by us) on the BONUS disk. OPEXEC and OPBROWSE are based on units in Turbo Analyst (EXECSWAP and BROWSE). The other units in the list were written from scratch.

And these are the new demonstration programs:

```
BROWSE
DESKPOP
DPINST
EDITOR
FORMTEST
POPHELP
PSCREEN
MAKESCRN
```

POPHELP is based on GUIDE, the on-line help facility for Turbo Professional that was added in version 5.07. Unlike GUIDE, however, POPHELP comes with source code, and thus can be considered a demonstration program. PSCREEN is a revised version of the program of the same name that appeared on the BONUS disk. Except for certain parts of DESKPOP, the others are brand new.

Those are the most notable changes in the package, but if you skim through the reference sections of the manuals you'll surely find lots of new things throughout the package. And please be sure to reread the overviews at the start of each reference section. In a few cases the documentation is virtually unchanged, but in other cases there is new and important information concerning the use of a particular unit.

B. Unit Dependencies

The following is a list of all of the units in the **Object Professional** library and the lower level units that they depend on, if any. Units that may be overlaid using Turbo Pascal's overlay system are marked with an asterisk ('*'). The meanings of the raised numbers that follow some of the unit names are given in the Legend that follows the list. For a list of the source files that make up a given unit, consult the file dependencies section of OPRO.MAK.

```
OP8087
   no dependencies
OPABSFLD *
   DOS, OPINLINE, OPSTRING, OPROOT, OPDATE[3], OPBCD[4]
OPASCIIZ *
   OPSTRING
OPBCD *
   no dependencies
OPBROWSE *
   DOS, OPINLINE, OPSTRING, OPROOT, OPCRT, OPMOUSE[2], OPCMD, OPDOS,
   OPFRAME, OPWINDOW
OPCLONE *
   DOS, OPINLINE, OPSTRING, OPROOT
OPCMD
   OPINLINE, OPCRT, OPMOUSE[2], OPROOT
OPCOLOR
   no dependencies
OPCRT
   DOS, CRT[1], OPINLINE
OPDATE *
   DOS, OPINLINE, OPSTRING
OPDEVICE *
   DOS, OPINLINE, OPSTRING, OPROOT, OPDOS, OPPRNLOW, OPPRINT
OPDIR *
   DOS, OPINLINE, OPSTRING, OPDOS, OPROOT, OPCRT, OPMOUSE[2], OPDATE[3],
   OPCMD, OPFRAME, OPWINDOW, OPPICK
OPDOS *
   DOS, OPSTRING
OPEDIT *
   OPINLINE, OPSTRING, OPROOT, OPCRT, OPMOUSE[2], OPCMD, OPDATE[3],
   OPBCD[4], OPABSFLD, OPFEDIT, OPFIELD
OPEDITOR *
   DOS, OPINLINE, OPSTRING, OPDOS, OPROOT, OPCRT, OPMOUSE[2], OPCMD,
   OPFRAME, OPWINDOW, OPMEMO
OPEMS
   no dependencies
OPENHKBD
   no dependencies
OPENTRY *
   OPINLINE, OPSTRING, OPROOT, OPCRT, OPMOUSE[2], OPCMD, OPDATE[3], OPBCD[4],
OPABSFLD,
   OPFEDIT, OPFIELD, OPFRAME, OPWINDOW, OPPICK[5], OPSELECT
OPERRHAN
   DOS
```

OPEXEC
   DOS

OPEXTMEM *
   no dependencies

OPFEDIT *
   OPINLINE, OPSTRING, OPROOT, OPCRT, OPMOUSE[2], OPDATE[3], OPBCD[4],
   OPCMD, OPABSFLD, OPFIELD[6]

OPFIELD *
   OPINLINE, OPSTRING, OPROOT, OPCRT, OPMOUSE[2], OPDATE[3], OPBCD[4],
   OPABSFLD, OPCMD, OPFEDIT[6]

OPFORM *
   DOS, OPINLINE, OPSTRING, OPROOT, OPDOS, OPDATE[3], OPBCD[4], OPABSFLD,
   OPPRNLOW, OPDEVICE, OPPRINT

OPFRAME *
   OPINLINE, OPSTRING, OPROOT, OPCRT, OPMOUSE[2], OPCMD

OPHELP *
   DOS, OPINLINE, OPSTRING, OPROOT, OPDOS, OPCRT, OPCMD, OPFRAME,
   OPWINDOW, OPMOUSE[2], OPPICK

OPINLINE
   no dependencies

OPINT
   DOS

OPLARRAY *
   DOS, OPINLINE, OPSTRING, OPDOS, OPROOT, OPEMS

OPMACED *
   DOS, OPINLINE, OPCRT, OPMOUSE[2], OPROOT, OPCMD, OPINT, OPMACRO

OPMACRO
   DOS, OPINT

OPMEMO *
   DOS, OPINLINE, OPSTRING, OPROOT, OPCRT, OPMOUSE[2], OPCMD, OPFRAME, OPWINDOW

OPMENU *
   OPINLINE, OPSTRING, OPROOT, OPCRT, OPMOUSE[2], OPCMD, OPFRAME, OPWINDOW

OPMOUSE
   OPCRT

OPPICK *
   OPINLINE, OPSTRING, OPROOT, OPCRT, OPMOUSE[2], OPCMD, OPFRAME, OPWINDOW

OPPRINT *
   DOS, OPINLINE, OPSTRING, OPROOT, OPDOS, OPPRNLOW

OPPRNLOW
   DOS, OPINLINE, OPROOT, OPDOS

OPREPLAY
   no dependencies

OPROOT
   DOS, OPINLINE, OPSTRING

OPSEDIT *
   OPINLINE, OPSTRING, OPROOT, OPCRT, OPMOUSE[2], OPCMD

OPSELECT *
   DOS, OPINLINE, OPSTRING, OPROOT, OPCRT, OPMOUSE[2], OPCMD, OPABSFLD, OPFIELD,
   OPFRAME, OPWINDOW

OPSORT *
   no dependencies

```
OPSTRDEV
   no dependencies
OPSTRING
   no dependencies
OPSWAP
   OPSWAP1
OPSWAP1
   DOS, OPINLINE
OPTSR
   DOS, OPINLINE, OPINT
OPWINDOW *
   OPINLINE, OPSTRING, OPROOT, OPCRT, OPMOUSE[2], OPCMD, OPFRAME
```

Bonus units:

```
OPCAL *
   OPINLINE, OPSTRING, OPROOT, OPCRT, OPMOUSE[2], OPCMD, OPFRAME,
   OPWINDOW, OPPICK, OPDATE
OPCALC *
   DOS, OPBCD, OPINLINE, OPSTRING, OPROOT, OPCRT, OPCMD, OPMOUSE[2],
   OPFRAME, OPWINDOW
OPQKREF *
   DOS, OPSTRING, OPINLINE, OPROOT, OPCRT, OPMOUSE[2], OPCMD, OPFRAME, OPWINDOW
```

Legend

1. {$IFDEF UseCrt}
2. {$IFDEF UseMouse}
3. {$IFDEF UseDates}
4. {$IFDEF UseBCD}
5. {$IFDEF PickListFields}
6. Circular dependency

C. Object Hierarchy

The following is a list of objects in the **Object Professional** library, and the objects they are derived from. Only two objects--ColorSet and Root--have no ancestors at all. All other objects are derived from Root. Indentation indicates that an object is derived from the next object above it at a reduced level of indentation. The unit in which each object is declared is given in the right hand column. The meanings of the raised numbers that follow some of the object names are given in the Legend that follows the list.

Objects marked with an asterisk ('*') are considered to be for internal use and are not documented in detail; these objects are typically accessed via methods of higher level objects. The field objects (StringField..TimeField, StringPrintField..LineField) are documented in groups in their respective units.

ColorSet	OPCRT
Root	OPROOT
PointerStack	"
WindowStack	OPWINDOW
ByteStack*	OPPRINT
StaticQueue	OPROOT
SingleListNode	OPROOT
TextField	OPFIELD
PrintMode	OPPRINT
FieldListNode*	OPFORM
PrintBlock	"
PrintPage	"
PrintedForm	"
DoubleListNode	OPROOT
HeaderNode*	OPFRAME
ShadowNode*[1]	"
HotNode*[2]	"
MenuItemNode	OPMENU
AbstractField	OPABSFLD
SelectField	OPFIELD
EntryField	"
StringField	"
ArrayField	"
CharField	"
BooleanField	"
YesNoField	"
LongIntField	"
WordField	"
IntegerField	"
ByteField	"
ShortIntField	"
RealField	"
BcdField[4]	"
ExtendedField[5]	"
DoubleField[5]	"
SingleField[5]	"
CompField[5]	"
DateField[6]	"
DateStField[6]	"
TimeField[6]	"
ChoiceField*	"
NestedField*	"
MultiLineField*	"
PickField*[7]	OPENTRY

PrintField	OPFORM
StringPrintField	"
CharPrintField	"
WordPrintField	"
BooleanPrintField	"
YesNoPrintField	"
LongPrintField	"
IntPrintField	"
BytePrintField	"
ShortPrintField	"
RealPrintField	"
BCDPrintField[4]	"
ExtPrintField[5]	"
DblPrintField[5]	"
SglPrintField[5]	"
CompPrintField[5]	"
DatePrintField[6]	"
TimePrintField[6]	"
LineField	"
ShadedField	"
SingleList	OPROOT
DoubleList	OPROOT
CircularList	"
MenuItemList*	OPMENU
BitSet	OPROOT
LargeBitSet	"
StringDict	"
StringSet	"
StringArray	"
AbstractArray	OPLARRAY
RamArray	"
RamRCArray	"
EmsArray	"
EmsRCArray	"
VirtualArray	"
PrimitiveArray	"
VirtualRCArray	"
OpArray	"
DirEntry	OPROOT
IdStream	"
DosIdStream	"
BufIdStream	"
Library	"
MemIdStream	"
MemLibrary	"
CommandProcessor	OPCMD
CommandPacker	"
Cloner	OPCLONE
ScrollBar*[3]	OPFRAME
ScreenRect	"
LoadableColorSet	OPWINDOW
PackedWindow	"
VirtScreen	"
LineEditor	OPEDIT
SimpleLineEditor	OPSEDIT
AbstractFrame*	OPFRAME
Frame	"
AbstractWindow*	OPWINDOW
RawWindow	"

SubMenu*	OPMENU
StackWindow	OPWINDOW
CommandWindow	"
Browser	OPBROWSE
Memo	OPMEMO
MemoFile	"
TextEditor	OPEDITOR
Menu	OPMENU
PickList	OPPICK
DirList	OPDIR
PathList	"
AbstractHelpWindow	OPHELP
PagedHelpWindow	"
ScrollingHelpWindow	"
AbstractSelector	OPSELECT
Selector	"
ScrollingSelector	"
EntryScreen	OPENTRY
ScrollingEntryScreen	"
BasePrinter	OPPRNLOW
DosPrinter	"
FlexiblePrinter	"
BiosPrinter	"
Printer	OPPRINT
Report	OPFORM

Legend

The indicated objects are declared only if the following conditional defines have been activated.

1. {$IFDEF UseShadows}
2. {$IFDEF UseHotSpots}
3. {$IFDEF UseScrollBars}
4. {$IFDEF UseBCD}
5. {$IFOPT N+}
6. {$IFDEF UseDates}
7. {$IFDEF PickListFields}

D. Error Codes

Object Professional uses a consistent set of rules throughout the library for defining and reporting error codes. A complete description of this system is provided with OPROOT (Chapter 8, section B.8), where the associated constants are declared. Here's a brief reminder about how error codes are defined:

Error codes are stored in variables of type Word. Each code is computed as the sum of two factors: an error prefix, which is always a multiple of 10000; and an error number, which is always less than 10000. An error "type" is defined as the error prefix divided by 10000. Hence, given an error Code, you can determine its type and number with the following statements:

```
ErrorType   := Code div 10000;
ErrorNumber := Code mod 10000;
```

or, putting the Code back together again:

```
Code := (10000 * ErrorType) + ErrorNumber;
```

The error type should be used to determine the program's response to the error, i.e., whether to halt, prompt for a fix, warn and retry, etc. See Chapter 8, section B.8 for more information. There are five error types and prefixes.

```
{Error types}
etFatal           = 0;  {fatal errors}
etNonFatal        = 1;  {non-fatal I/O errors}
etWarning         = 2;  {warnings, user input errors, etc.}
etMessage         = 3;  {simple messages}
etNoError         = 4;  {no error}

{Error prefixes}
epFatal           = etFatal    * 10000;
epNonFatal        = etNonFatal * 10000;
epWarning         = etWarning  * 10000;
epMessage         = etMessage  * 10000;
```

Error numbers provide more detail about an error, usually enough to determine why the error occurred. The lowest error numbers, ranging from 2 (file not found) to 162 (general failure), are the same as Turbo Pascal's I/O error codes. Other numbers are organized loosely in blocks of 1000, with each block indicating a different category of error:

```
0000-0999   DOS, Turbo Pascal, DOS critical errors
1000-1999   Other I/O errors, including stream-related errors
2000-2999   Capacity or environment errors
7000-7999   Warnings, user errors
8000-8999   Programming errors
9000-9999   Messages
```

The following is a complete list of all the error numbers. If you get one of these error numbers following a call to a particular object's method, see the documentation for that method for more details. We strongly recommend that you refer to error numbers by name rather than by a specific numeric value.

DOS, Turbo Pascal, and DOS Critical Errors

```
{DOS errors}
ecFileNotFound    = 00002; {file not found}
ecPathNotFound    = 00003; {path not found}
ecTooManyFiles    = 00004; {too many open files}
```

```
ecAccessDenied    = 00005; {file access denied}
ecInvalidHandle   = 00006; {invalid file handle}
ecOutOfMemory     = 00008; {insufficient memory}
ecInvalidDrive    = 00015; {invalid drive}
ecNoMoreFiles     = 00018; {no more files}

{Turbo Pascal I/O errors}
ecDiskRead        = 00100; {attempt to read beyond end of file}
ecDiskFull        = 00101; {disk is full}
ecNotAssigned     = 00102; {file not Assign-ed}
ecNotOpen         = 00103; {file not open}
ecNotOpenInput    = 00104; {file not open for input}
ecNotOpenOutput   = 00105; {file not open for output}
ecInvalidFormat   = 00106; {invalid format for packed window}

{DOS critical errors}
ecWriteProtected  = 00150; {disk is write-protected}
ecUnknownUnit     = 00151; {unknown disk unit}
ecDriveNotReady   = 00152; {drive is not ready}
ecUnknownCommand  = 00153; {unknown command}
ecCrcError        = 00154; {data error}
ecBadStructLen    = 00155; {bad request structure length}
ecSeekError       = 00156; {seek error}
ecUnknownMedia    = 00157; {unknown media type}
ecSectorNotFound  = 00158; {disk sector not found}
ecOutOfPaper      = 00159; {printer is out of paper}
ecDeviceWrite     = 00160; {device write error}
ecDeviceRead      = 00161; {device read error}
ecHardwareFailure = 00162; {general failure}
```

Other I/O Errors, including Stream-related Errors

```
{reported by OPBROWSE and OPEDITOR}
ecFileIsEmpty     = 01000; {file is empty}

{reported by OPROOT.IdStream}
ecIdUnknown       = 01101; {unknown object id read from stream}
ecLoadFailed      = 01102; {load constructor failed within Get or GetPtr}
ecInvalidData     = 01103; {attempt to read/write bad data}

{reported by OPROOT.Library}
ecNameBlank       = 01200; {can't add entry if name is blank}
ecPackFailed      = 01201; {error packing library}
ecBadSignature    = 01202; {bad signature in directory}
```

Capacity or Environment Errors

```
{reported by OPROOT.StaticQueue}
ecQueueFull       = 02000; {queue is full. new element rejected}
ecQueueEmpty      = 02001; {queue is empty. no element returned}

{reported by OPROOT.Library}
ecDirectoryFull   = 02020; {directory full--can't add entry}

{reported by OPCMD}
ecPartialMatch    = 02050; {cannot add command due to partial match}
ecKeyTableFull    = 02051; {cannot add command due to full key table}

{reported by OPLARRAY}
ecInsufficientEms = 02100; {insufficient EMS memory to allocate}
ecEmsAllocation   = 02101; {error allocating EMS page}
ecNoEms           = 02102; {EMS operation failed because no EMS available}
ecEmsPageMapping  = 02103; {error mapping EMS page}
```

```
ecCantFreeEms     = 02104; {cannot deallocate EMS handle}

{reported by OPDIR}
ecDirTooBig       = 02200; {too many files to fit into directory buffer}
```

Warnings and User Errors

```
{reported by OPFIELD, OPSELECT, OPENTRY}
ecFieldRequired   = 07000; {field is required}
ecBadFormat       = 07001; {bad format (number/date/time, etc.}
ecOutOfRange      = 07002; {value entered is out of range}
ecBlanksInField   = 07003; {field contains blanks}
ecPartialEntry    = 07004; {field contains a partial entry}
ecBadCharacter    = 07005; {field contains an illegal character}

{reported by OPMEMO, OPEDITOR, OPBROWSE}
ecStringNotFound  = 07100; {search string not found}
ecNotToScreen     = 07101; {cannot write output to screen}
ecInvalidNumber   = 07102; {invalid number entered}

{reported by OPMEMO, OPEDITOR}
ecBufferFull      = 07110; {edit buffer is full}
ecLineTooLong     = 07111; {line too long, CRLF inserted}
ecTooManyLines    = 07112; {max line limit would be exceeded}
ecOverLineLimit   = 07113; {max line limit already exceeded}
ecFileTruncated   = 07114; {file truncated}
ecFileTooLarge    = 07115; {file too large to fit in buffer}
ecNoBlock         = 07116; {block not marked or hidden}

{reported by OPCALC}
ecDivideByZero    = 07200; {divide overflow}
ecUndefinedNum    = 07201; {error returned by 80x87 coprocessor}

{reported by OPDIR}
ecNoPickSelection = 07300; {no pick selection made}

{reported by OPROOT.Library}
ecDuplicateName   = 07400; {name already in use}
ecEntryNotFound   = 07401; {library entry not found}
```

Programmer Errors

```
{reported by OPROOT.StringSet}
ecDupString       = 08000; {string is already in string set}

{reported by OPROOT.IdStream}
ecIdNotRegistered = 08010; {user object not registered with stream}
ecIdRegisteredDup = 08011; {attempt to register duplicate object id}

{reported by OPLARRAY}
ecElSizeIsZero    = 08050; {element size is zero}
ecIncorrectSize   = 08051; {incorrect dimensions for disk file}
ecRowOutOfRange   = 08052; {specified row is out of range}
ecColOutOfRange   = 08053; {specified column is out of range}
ecNotLArrayFile   = 08054; {specified file is not a OpLarray file}
ecElementTooBig   = 08055; {Element size is too big}
ecBadDimensions   = 08056; {an invalid dimension was specified}
ecLessThanOnePage = 08057; {less than one page in RAM}
ecFlushError      = 08058; {error flushing large array file}

{reported by OPWINDOW and others}
ecWinCoordsBad    = 08100; {bad coordinates specified for a window}
ecWinNotActive    = 08101; {window must be active for this operation}
ecWinNotCurrent   = 08102; {window must be current for this operation}
```

```
ecWinIsActive     = 08103; {window must be inactive for this operation}
ecWinInaccessible = 08104; {stacked window not accessible}
ecWinIsZoomed     = 08105; {zoomed window cannot be zoomed again}
ecWinBadIndex     = 08106; {invalid header or hot spot index}
ecWinNotSizeable  = 08107; {window must be sizeable for this operation}
ecNoProcessor     = 08108; {no command processor is available}

{reported by OPPICK}
ecNoPickItems     = 08200; {attempt to pick from empty list}
ecBadPickOrient   = 08201; {invalid pick orientation}
ecBadPickHandler  = 08202; {invalid pick command handler}

{reported by OPHELP}
ecHelpInvalid     = 08220; {invalid help file format}
ecNoHelpForTopic  = 08221; {no help for specified topic}
ecNoBoundHelp     = 08222; {pointer for bound-in help not registered}

{reported by OPSELECT, OPENTRY}
ecTooManyFields   = 08300; {too many fields in a selector/entry screen}
ecBadCoordinates  = 08301; {bad coordinates for a field}
ecNotScrollable   = 08302; {virtual screen not allocated}
ecNoFields        = 08303; {selector/entry screen has no fields}
ecFieldNotFound   = 08305; {attempt to position cursor on non-existent field}

{reported by OPPRINT}
ecInvalidPrinter  = 08400; {specified Printer type is invalid}
ecPosOutOfRange   = 08401; {printer position out of range}
ecNoPositionPrim  = 08402; {no printer position routine specified}
ecFontStkOverflow = 08403; {stack overflow in font stack}
ecFontStackError  = 08404; {stack underflow in font stack}
ecByteStkOverflow = 08405; {stack overflow in a byte stack}
ecNoPrnDriver     = 08406; {no BasePrinterPtr specified for Printer}

{reported by OPFORM}
ecNoFieldsOnPage  = 08420; {page to be printed has no active fields}
ecNoBlockDefined  = 08421; {attempt to add field when no block defined}
ecNoSuchField     = 08422; {FieldID represents non-existent field}
ecDiagNotAllowed  = 08423; {diagonal lines not allowed}

{reported by various units}
ecNilPointer      = 08500; {nil pointer where there shouldn't be one}
ecBadParam        = 08501; {bad parameter to a procedure}
```

Messages (not errors)

```
mcLineNumber      = 09000; {prompt for line number}
mcOptionString    = 09001; {prompt for search options}
mcBlockWrite      = 09002; {prompt for file to write marked block to}
mcBlockRead       = 09003; {prompt for file to read in as marked block}
mcNewFile         = 09004; {prompt for new file name}
mcRightMargin     = 09005; {prompt for a right margin}
mcIndentLevel     = 09006; {prompt for an indent level}
mcTabSize         = 09007; {prompt for a tab size}
mcFileModified    = 09008; {"file modified--save it?" prompt}
mcSaveAs          = 09009; {new name for a file}
mcSearchFor       = 09010; {prompt for a search string}
mcReplaceWith     = 09011; {prompt for a replacement string}
mcReplaceYesNo    = 09012; {prompt to confirm a replacement}
```

E. Using Overlays with Object Professional

When writing overlaid programs with **Object Professional**, there are only a few issues that you need to keep in mind, but they are important ones. First and foremost, not all of the units in **Object Professional** may be overlaid. The following is a list of the units that you *can* overlay:

```
OPABSFLD
OPASCIIZ
OPBCD
OPBROWSE *
OPCAL *
OPCALC *
OPCLONE
OPDATE *
OPDEVICE
OPDIR
OPDOS
OPEDIT *
OPEDITOR *
OPENTRY *
OPEXTMEM *
OPFEDIT
OPFIELD
OPFORM
OPFRAME
OPHELP *
OPLARRAY *
OPMACED *
OPMEMO *
OPMENU *
OPPICK *
OPPRINT *
OPQKREF *
OPSEDIT *
OPSELECT *
OPSORT
OPWINDOW
```

An asterisk ('*') beside the name of a unit indicates that the unit has an initialization block. If you overlay one of these units, your program *must* initialize the overlay manager before that unit's initialization code is executed. See the discussion of "Initialization Sections in Overlaid Units" in the Turbo Pascal Reference Guide (p. 178) for instructions on how to do this.

Keep in mind, however, that by default *all* **Object Professional** units have initialization blocks, not just the ones marked with asterisks. Those that don't have any actual initialization code will have empty initialization blocks if the conditional compilation directive InitAllUnits is defined in OPDEFINE.INC. For an explanation of why this directive is defined by default, see the discussion of OPDEFINE.INC in Chapter 2, "Principles of Organization."

An even more important issue to bear in mind is that of performance. Many of the units that appear in this list depend on one another, and you must be careful not to create situations in which two units are constantly swapped in and out of memory because one depends on the other--a phenomenon called "overlay thrashing." (A complete list of unit dependencies appears in Appendix A, above.) In general, you should not overlay a unit that depends on another overlaid unit unless you can insure that both units can be kept in memory while the dependent unit is loaded and active. Sometimes this can be accomplished just by increasing the size of

your overlay buffer, but not always. In most cases, when you have to choose, it is better to overlay the dependent unit than the one it depends on.

A subtler issue to keep in mind, at least in some programs, is that of window stacks (see the documentation for OPWINDOW). Consider the case of DESKPOP. If you've experimented with it, you've probably wondered why we didn't use overlays to limit its memory usage and free up additional heap space. It's a fair question, and the answer has to do with window stacks. When you have multiple windows of various kinds on a stack, the stack manager will sometimes have to call each window's UpdateContents method in order to redraw it. If you have six window-based objects of different kinds on the stack, and all the objects's UpdateContents methods are implemented in overlayed units, then this seemingly simple operation can potentially cause all six units to be reloaded from disk in succession. As a result, screen updates in such situations would be unacceptably slow. DESKPOP is perhaps an extreme case, but the same issue can arise in smaller, simpler programs with fewer windows as well.

All we can do here is give you some basic guidelines and rules of thumb. Only through experimentation will you be able to determine what units should and should not be overlaid in a given program. When a program starts to get too large, many programmers look to overlays to provide a quick fix for the problem and consequently rush the conversion process. It rarely works. Before converting a program to use overlays, do your homework.

Formulate a complete list of all your unit dependencies, and generate a MAP file for your program ("TPC filename.ext /GS") to see which of the units that can be overlaid use the most memory. Using this information, make a second list of the units that seem like good candidates for overlaying. Then spend some time--as much as you can spare--experimenting with various combinations. If you are running your experiments with the overlays stored in EMS memory, and not all your users will have EMS in their systems, be sure to test your program with overlays stored on disk as well, preferably on an 8088-based machine. Running all your tests on a fast '386 machine with the overlays loaded into EMS memory can produce misleading results.

Above all, be prepared for the possibility that overlays may not be the best solution for a particular program's memory problems. In some cases, you may find that a better solution is to break a large program into a series of smaller programs executed by a shell program with a main menu. If you decide to take that approach instead, we suggest that you look at both the ExecDos function in OPDOS and the ExecDosSwap function in OPEXEC. ExecDos is often a good choice if the shell program is relatively small. ExecDosSwap is a better choice if the shell program is large, or memory usage needs to be kept to an absolute minimum, or the programs being executed need all the memory they can get.

F. Converting Turbo Professional Applications

The most honest advice we can provide for this section is: don't attempt to convert applications that rely heavily on the user-interface capabilities of Turbo Professional. If you'd like to take advantage of **Object Professional**'s capabilities in an existing application, use this opportunity to rewrite it from the ground up. A semi-mechanical conversion won't provide the full benefits of either an object-oriented design or the new features of **Object Professional**.

That said, we'll try to back off and provide some guidelines for conversion. Understand, however, that this is not the kind of conversion in which you can substitute one keyword for another, or even do it in small steps. Object-oriented programming is a different paradigm; to take full advantage of it, you need to change your thinking. We've been quite surprised at the number of recent phone conversations that began: is your **Object Professional** package source-compatible with Turbo Professional? If you have that question in the back of your mind still, here's the blunt answer: where user-interface capabilities are concerned, No.

As we've noted already in Appendix A, many of the non-object-oriented units in **Object Professional** are quite similar, if not identical, to those in Turbo Professional. If your application relies only on the following units, you should be able to adjust your USES statement and recompile: OP8087, OPASCIIZ, OPBCD, OPCRT, OPDATE, OPDOS, OPEMS, OPENHKBD, OPERRHAN, OPEXTMEM, OPINLINE, OPINT, OPMACED, OPMACRO, OPMOUSE, OPSORT, OPSTRDEV, OPSTRING, OPTSR. One change to note: the TerminateAndStayResident function in OPTSR has been changed to a procedure and renamed to StayRes (because the name was so long that we were always tripping over it).

Applications that use other Turbo Professional units now rewritten for **Object Professional**--OPDIR, OPEDIT, OPENTRY, OPHELP, OPMEMO, OPMENU, OPPICK, OPWINDOW--will require more work.

As an object-oriented language goes, Turbo Pascal 5.5 is very easy to approach. You *can* mix object-oriented capabilities with traditional structured programming. The object-oriented extensions use a familiar record-like syntax. Similarly, as an object-oriented library is measured, **Object Professional** should be easy to comprehend, especially for users of Turbo Professional. We set the following guidelines for ourselves to make your transition as easy as possible:

- we wouldn't change the system-oriented units
- we would provide the same high level capabilities (e.g., windows, pick lists, entry screens)
- we would minimize non-essential abstraction
- we would avoid an event-driven architecture

The first two guidelines should already be clear. The third is related but still distinct from the first two. If you've studied other object-oriented languages or class libraries, you'll know that some of them go overboard with abstraction--there are often several layers in the class hierarchy that not only serve incomprehensible purposes but also reduce the efficiency of basic operations, sometimes as basic as addition and subtraction. On the other hand, we can see the advantages of abstraction for increasing the extensibility and reusability of certain objects. We've tried to maintain a balance; abstract objects such as AbstractWindow, AbstractSelector, and AbstractHelpWindow were used only after prolonged debate.

The final guideline particularly sets **Object Professional** apart from operating environments like the Macintosh operating system, Microsoft Windows, and Presentation Manager, all of which are event-driven. After hearing of the steep learning curve for these environments, we felt justified in sticking with our Process loops and background task handlers.

If OOP is new to you, we recommend that you start out by converting a smaller application rather than a large one. A good exercise would be to convert WINWOW from Turbo Professional to **Object Professional** (and then check how similar your result is to our own **Object Professional** version).

Some changes made for **Object Professional** weren't driven at all by object-oriented programming. Turbo Professional had gone through three major generations of enhancements and the "room additions" were starting to overbalance the original "foundation." Knowing that the transition to OOP would break most programs anyway, we took the opportunity to rebuild our own foundations. Here are some of the goals we set:

- slash the use of global typed constants for determining unit behavior (particular culprits were TPEDIT and TPPICK)
- define a comprehensive set of video attributes that could apply to an entire application
- treat all window types (windows, pick lists, menus, entry screens, etc.) consistently
- assure that all video operations could occur on a virtual screen as well as on the physical screen

The remainder of this section will offer specific advice for the Turbo Professional units requiring the most conversion effort. To save space, we'll refer to Turbo Professional as TPRO, and **Object Professional** as OPRO.

TPCRT -> OPCRT

OPCRT is almost identical to TPCRT. The only significant difference is that all of the window-relative screen writing routines in TPCRT (FastWriteWindow, FastReadWindow, etc.) have been incorporated into OPWINDOW instead.

The ColorSet object defined by OPCRT is used by all other OPRO user-interface objects to determine colors. TPRO color management was not so consistent: some routines took a series of video attributes as parameters; others took a single, special-purpose array of video attributes as a parameter; others required global variables to be modified; and so on. One of your first conversion tasks will be to survey all your colors and to build one (or perhaps more) ColorSets.

TPCMD -> OPCMD

This transition will require effort only if you've taken advantage of the features described in Appendix C of the TPRO manual, "Keyboard Handling in Turbo Professional." The following OPRO CommandProcessor methods are appropriate for conversion:

Category	TPRO	OPRO
Installable Keyboard Hooks	AddEditCommand, etc.	AddCommand
Background Task Hooks	EditKeyPtr, etc.	SetGetKeyProc
Context-Sensitive Help Hooks	EditHelpPtr, etc.	SetHelpProc

TPWINDOW -> OPFRAME, OPWINDOW

Understanding the differences between TPWINDOW and OPWINDOW is the key to approaching OPRO. Because almost all of the remaining user interface objects are derived from OPRO's RawWindow type, what you learn about RawWindow will apply to all the other objects as well.

In order to tap OPRO's powerful new windowing capabilities--multiple headers, generalized shadows, inheritable scroll bars on any border edge, etc.--the simple TPRO MakeWindow call

must be replaced by a series of calls to OPWINDOW and OPFRAME methods. The following tables shows the correspondence:

TPRO MakeWindow Parameter	OPRO Equivalent
var W : WindowPtr	RawWindow, StackWindow, etc. instance variable
XLow, YLow, XHigh, YHigh : Byte	Init, InitCustom coordinates*
DrawFrame : Boolean	wBordered option to InitCustom or wOptionsOn
ClearWindow : Boolean	wClear "
SaveWindow : Boolean	wSaveContents "
WindowAttr : Byte	TextColor, TextMono fields in ColorSet
FrameAttr : Byte	FrameColor, FrameMono "
HeaderAttr : Byte	HeaderColor, HeaderMono "
Header : String	Frame.AddHeader and related methods
TPRO TPWINDOW Typed Constant	OPRO Equivalent
Shadow : Boolean	Frame.AddShadow and related methods
ShadowMode : ShadowType	AddShadow DrawType parameter
ShadowAttr : Byte	ShadowColor, ShadowMono fields in ColorSet
Explode : Boolean	RawWindow.EnableExplosions
ExplodeDelay : Word	EnableExplosions StageDelay parameter
SoundFlagW : Boolean	wSoundEffects option to InitCustom or wOptionsOn

* OPRO always uses the active window coordinates, not counting the border, whereas TPRO's coordinates included the border.

We recommend that you replace TPRO WindowPtr variables with OPRO instance variables of the WindowPtr type. In this way your window objects will be pointers to StackWindow, allowing you to randomly access all windows on the screen. Here's the equivalent code for making, displaying, writing to, and erasing a full-screen bordered window in TPRO and OPRO:

```
program TPROexample;
uses TpCrt, TpWindow;
var   W : WindowPtr;
begin
  Shadow := True;
  Explode := True;
  if not MakeWindow(W, 1, 1, ScreenWidth, ScreenHeight,
                    True, True, True, $07, $0F, $0F,
                    'Full Screen Window') then Halt;
  if not DisplayWindow(W) then Halt;
  FastWriteWindow('top left corner of active window', 1, 1, $07);
  ReadLn;
  DisposeWindow(EraseTopWindow);
end.
```

```
program OPROexample;
uses OpCrt, OpRoot, OpFrame, OpWindow;
var  W : WindowPtr;
begin
  New(W, InitCustom(2, 2, ScreenWidth-1, ScreenHeight-1,
                    DefaultColorSet, wBordered+wClear+wSaveContents));
  if W = nil then
    Halt;
  W^.wFrame.AddHeader('Full Screen Window', heTC);
  W^.wFrame.AddShadow(shBR, shSeeThru);
  W^.EnableExplosions(15);
  W^.Draw;
  W^.wFastWrite('top left corner of active window', 1, 1, $07);
  ReadLn;
  W^.Erase;
  W^.Done;
end.
```

TPMENU -> OPMENU

Except for syntactic differences associated with OOP and some name changes, OPMENU is very similar to TPMENU. The following table shows that there is a direct mapping between the routines for building a menu system:

TPRO	OPRO
NewMenu	Init or InitCustom
SubMenu	AddSubMenu or AddFramedSubMenu
PopSubLevel	ItemsDone
MenuItem	AddItem or AddItemHelp
MenuItemPtr	AddItemPtr or AddItemHelpPtr

Many of the routines that had been tacked on to TPMENU for customizing menu systems--MenuWidth, MenuHeight, MenuMode, etc.--are not needed in OPRO because the features are inherited from the window objects.

The best way to convert an existing menu system is to use MAKEMENU to re-enter it and generate the source code.

Another difference to consider is the way OPRO gets a menu choice from the user. Here's a short program from each system:

```
program TPROexample;
uses TpCrt, TpWindow, TpMenu;
var
  Ch : Char;
  Main : Menu;
  Key : MenuKey;
procedure InitMenu(var M : Menu);
begin
  ...
end;
begin
  InitMenu(Main);
  Key := MenuChoice(Main, Ch);
  if MenuCmdNum = MKSSelect then
    case Key of
      1 : {process item 1}
      {...other menu items...}
    end;
end.
```

```
program OPROexample;
uses OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpMenu;
var  Main : Menu;
function InitMenu(var M : Menu) : Word;
begin
  ...
end;
begin
  if InitMenu(Main) <> 0 then
    Halt;
  Main.Process;
  if Main.GetLastCommand = ccSelect then
    case Main.MenuChoice of
      1 : {process item 1};
      {...other menu items...}
    end;
end.
```

TPEDIT -> OPSEDIT or OPEDIT

Calls to TPEDIT.ReadString may be converted to use either of two OPRO objects: the SimpleLineEditor in OPSEDIT, or the LineEditor in OPEDIT. SimpleLineEditor's ReadString method closely matches TPEDIT's procedure and has the advantage of generating less executable code than OPEDIT. OPEDIT provides many more features, including optional picture mask support.

Calls to TPEDIT procedures like ReadInteger, ReadCharacter, and YesOrNo must be converted to use OPEDIT's LineEditor, since there are no equivalent methods in SimpleLineEditor. Also note that if you're using the entry screen system (OPENTRY), you may as well use OPEDIT, since they share much of the same low-level code (in OPABSFLD, OPFEDIT, and OPFIELD).

OPEDIT and OPSEDIT follow OPRO's general dictum of avoiding global typed constants to specify behavior. The following short programs show what we mean:

```
{$V-}
program TPROexample;
uses
  TpCrt, TpEdit;
var
  S : String[80];
  Escaped : Boolean;
begin
  ForceUpper := True;
  WindowRelative := True;
  EditSize := 10;
  S := '';
  ReadString('Enter string: ', 1, 1, 80, $0F, $70, $70, Escaped, S);
  if not Escaped then
    WriteLn(^M^J'You entered ', S);
end.
```

```
{$V-}
program OPROexample;
uses
  OpCrt, OpCmd, OpSEdit;
var
  SE : SimpleLineEditor;
  S : String[80];
begin
  SE.Init(DefaultColorSet);
  SE.seOptionsOn(seForceUpper+seWindowRelative);
  S := '';
  SE.ReadString('Enter string:', 1, 1, 80, 10, S);
  if SE.GetLastCommand = ccSelect then
    WriteLn(^M^J'You entered ', S);
end.
```

TPHELP -> OPHELP

Help text files are upwardly compatible between TPRO and OPRO. Just use the OPRO version of MAKEHELP to recompile the help text into a new binary file. Additional conversion requirements are largely limited to the differences in the underlying window and command processing systems. The following comparative examples use the ENTRY demonstration program's help file:

```
program TPROexample;
uses
  TpCrt, TpHelp;
const
  HelpColors : HelpColorArray = ($07, $07, $0F, $70, $09, $70, $0F, $01);
var
  Help : HelpPtr;
  Topic : Word;
begin
  if OpenHelpFile('ENTRY.HLP', 8, 7, 19, 2, HelpColors, Help) <> 0 then Halt;
  repeat
    Topic := PickHelp(Help, 2, 2, 10, 1);
    if Topic <> 0 then
      if not ShowHelp(Help, Topic) then
        WriteLn('Error showing help');
  until Topic = 0;
  CloseHelp(Help);
end.
```

```
program OPROexample;
uses
  OpCrt, OpCmd, OpFrame, OpWindow, OpPick, OpHelp;
var
  Help : PagedHelpWindow;
  Topic : Word;
begin
  if not Help.InitCustom(9, 8, 72, 18, DefaultColorSet,
                         DefWindowOptions or wBordered,
                         'ENTRY.HLP', PickVertical) then Halt;
  Help.AddTopicHeader(1, 60, heTC);
  Help.AddMoreHeader(' || for more ', heBR, #24, #25, '', 2, 3, 0);
  Help.AddMoreHelpHeader(' PgUp/PgDn ', heBR, 'PgUp', 'PgDn', '/', 2, 7, 6);
  repeat
    Help.SetTopic(PickTopic);
    Help.Process;
    if Help.GetLastCommand = ccSelect then
      Topic := Help.GetTopicChoice
    else
      Topic := 0;
    if Topic <> 0 then begin
      Help.SetTopic(Topic);
      Help.Process;
      if Help.GetLastCommand = ccError then
        WriteLn('Error showing help');
    end;
  until Topic = 0;
  Help.Done;
end.
```

TPPICK -> OPPICK

As with TPHELP, conversion requirements for TPPICK are largely limited to the differences in the underlying window and command processing systems. OPPICK provides one of the best opportunities for you to derive your own object types, however, and we hope you'll take advantage of it. The bonus unit OPCAL provides a good example; the OPPICK chapter lists others.

TPDIR -> OPDIR

OPDIR generalizes upon TPDIR by providing many more direct hooks into the PickList object from which it is derived. Again, once you've understood OPRO's window and command processing systems, you should have little trouble making this conversion.

TPENTRY -> OPENTRY

OPENTRY has all of the same features as TPENTRY--multiple field types, scrolling entry screens, pre- and post-edit hooks, validation routines, protected fields, and so on. We recommend that you use the new MAKESCRN program to design and generate source code for entry screens that look like your old ones. Then, if you've taken advantage of user-written hooks that MAKESCRN can't write for you, fill in the shell routines MAKESCRN does generate with code that should be very similar to the TPRO version.

An excellent source of information about the TPENTRY to OPENTRY conversion is the ENTRY demonstration program. The TPRO and OPRO versions perform almost identical functions and both of them exercise the majority of the capabilities of the entry system.

Note that the OPENTRY system does count on more units than did TPENTRY:

TPENTRY

```
uses
  TpInline, TpMemChk, TpCrt,
  {$IFDEF UseMouse} TpMouse, {$ENDIF}
  {$IFDEF UseBCD} TpBcd, {$ENDIF}
  TpCmd, TpString, TpWindow, TpDate;
```

OPENTRY

```
uses
  OpInline, OpString, OpRoot, OpCrt,
  {$IFDEF UseMouse} OpMouse, {$ENDIF}
  {$IFDEF UseBcd} OpBcd, {$ENDIF}
  {$IFDEF UseDates} OpDate, {$ENDIF}
  OpCmd, OpAbsFld, OpFEdit, OpField, OpFrame, OpWindow,
  {$IFDEF PickListFields} OpPick, {$ENDIF}
  OpSelect;
```

Some of this expansion in unit count occurs because the EntryScreen object is derived from a CommandWindow. One of the most common questions about TPENTRY was "how can I save and restore the screen contents underneath my entry screen?" That question, and many others relating to window borders, headers, resizeability, and so on, are answered automatically now because the EntryScreen inherits all of CommandWindow's features.

The second factor causing more units was the unification of the field entry system. Now, OPEDIT (the LineEditor object) shares line editors with OPENTRY via the OPABSFLD, OPFEDIT, and OPFIELD units. OPFORM (the PrintedForm object) shares picture mask interpretation code with OPENTRY via the OPABSFLD unit.

G. Compiling Large Programs

One of the most frustrating error messages to receive during a compilation is the dreaded "Error 1: Out of Memory" message. It usually means you have to interrupt the development process and make some (occasionally drastic) modifications to your environment and/or your programs. At best, this is an inconvenience; at worst it can mean rethinking the fundamental application structure.

When taking advantage of all that **Object Professional** has to offer, some of your large programs may run into this limit. Since this is a fairly common technical support question, we've listed some of the standard steps to take to maximize Turbo Pascal's use of memory. Some of these are listed in the Turbo Pascal manual, some are the results of our (and others') experience:

1. In the Integrated Development Environment (TURBO.EXE), set the compile destination to Disk instead of Memory (Compile-Destination-Disk).
2. Set the compiler's link buffer to disk. In the IDE, set Options-Compiler-Link to Disk. Using the command line compiler (TPC.EXE), specify the /L option.
3. If possible, turn off range and stack checking (use $R-,S- at the top of your programs and units). Remember, however, that the run-time library will no longer trap these errors for you (instead, your program will just overwrite memory somewhere). You'll probably want to leave these checks on in units under development. Never compile the **Object Professional** units with these checks turned on.
4. When debugging, enable debug information (with the $D+ and $L+ directives) only for those units that you expect to trace into. After you finish debugging a unit, don't forget to recompile it with debug information turned off.
5. Start with as much memory as possible by removing non-essential TSRs and device drivers.
6. Use TPC.EXE instead of TURBO.EXE.
7. Extract all units except PRINTER from TURBO.TPL (creating SYSTEM.TPU, DOS.TPU, etc. instead). Since the linker keeps all of TURBO.TPL in memory, this will free up some memory. You can even discard TURBO.TPL completely (as long as you have .TPU units for all the system units), though compiler will give you a warning message when you do this.
8. Keep your main program as small as possible. In the extreme, this would mean a single line of code--a call to another unit that really contains the "main block." For example:

```
program BigProg;     {file BIGPROG.PAS}
uses
  BPMain;
begin
  Main;
end.

unit BPMain;         {file BPMAIN.PAS}
interface
uses {.....};
  procedure Main;
    {-Main procedure for BigProg}

implementation
  procedure Main;
  begin
    {...}
  end;
end.
```

Here, the main program uses only one unit and has only one executable statement--a call to the "real" main block in the unit BPMain. Since the compiler keeps the main block in memory during compile and link, this reduces that memory requirement.

9. Move as many declarations as you can from the interface section of each unit to the implementation. Doing so reduces the amount of symbol information that the compiler needs to retain throughout the compilation cycle. In reference to item 8, note that all global declarations previously located in the main program could be moved to the implementation section of unit BPMain.

These suggestions are not necessarily listed in the order you would employ them. That depends on your environment and what you consider to be most convenient.

If you try all of these suggestions and your program is still too large to compile, then it's time to break it into several separate EXE files. These EXE files can transfer control between themselves by using the DOS Exec function. (Note that Turbo Pascal's overlay system, while it saves memory at runtime, does not help you to compile larger programs.) The OPEXEC unit provides a swapping Exec function that allows the executed program to reuse most of the memory space of its caller.

H. For Further Reading

If you need or want further background on the inner workings of PCs, MS-DOS, and/or the IBM BIOS, we recommend the following:

1. Peter Norton, *Programmer's Guide to the IBM PC* (Microsoft Press). A good general reference book covering all three topics.
2. Ray Duncan, *Advanced MS-DOS* (Microsoft Press). Long the best reference book on DOS and DOS function calls.
3. Terry R. Dettmann, *DOS Programmer's Reference* (Que). A little-known gold mine. Besides the basic stuff covered by Duncan and Norton, this one discusses undocumented DOS calls and other unusual topics such as mouse programming. Highly recommended.
4. Robert Jourdain, *Programmer's Problem Solver for the IBM PC, XT, and AT* (Brady Books). Not a reference book per se, but contains much useful background information.
5. Richard Wilton, *Programmer's Guide to PC and PS/2 Video Systems* (Microsoft Press). Gives very detailed information on the various display adapters.

Supplementary material about object-oriented programming in Turbo Pascal 5.5 is scarce at this time. A book has not yet been published that is destined to become a classic in the same sense as many of those listed above. Be sure that you carefully read Borland's OOP Guide for Turbo Pascal 5.5; it is the most easily approachable introduction to OOP that we've seen. You might also consider the following books:

1. Brad J. Cox, *Object-Oriented Programming: An Evolutionary Approach* (Addison-Wesley). A very readable introduction to OOP using the Objective-C language as a base.
2. Bertrand Meyer, *Object-Oriented Software Construction* (Prentice-Hall). Object-oriented programming with plenty of emphasis on the fundamental principles. Uses the Eiffel language (designed and sold by Meyer) as a base.
3. Kurt J. Schmucker, *Object-Oriented Programming for the Macintosh* (Hayden Books). Based on Apple's Object Pascal, which is similar to Turbo Pascal 5.5, but enough different that it may be misleading. The book is also fairly specific to dealing with the Macintosh operating systems.
4. Tom Swan, *Mastering Turbo Pascal 5.5* (Hayden Books). The newest edition of Tom's well-respected reference on Turbo Pascal. The 5.5 edition includes one chapter (60 pages) specifically about OOP. The chapter is nicely done, including a good discussion of streams.

If you find a book that deserves to be on this list, please let us know.

Procedure and Function Reference

This section provides a complete list of all documented procedures, functions, and methods in the entire library. The entry for each routine is of the following form:

```
(v) c-ppp      procedure ProcedureName(parameters);
                 {-Brief description}
```

The first column shows the volume (v), chapter (c), and page (ppp) where the routine is documented in the manual. The second column shows the name of the routine and the parameters it takes, followed by a brief description of the routine. In a very few cases, you will see an entry like this

```
(v) c-ppp      procedure ProcedureName(...);
                 {-Brief description}
```

where the parameters are shown as '...'. This is done (to save space) in a few cases where a routine has a lengthy parameter list that is identical to that of the previous routine.

The individual entries are grouped according to unit and object name. For example, the entries for OPCRT are arranged into two groups, one for regular procedures and functions, and one for the methods for the ColorSet object. A dividing line precedes each section, and just above the line is the name of the unit and, if the routines in the group are methods, the name of the object they belong to.

The arrangement of the entries within each group is not alphabetical; rather, it is based on the order in which the routines are declared in the source code for the unit/object. This order was chosen because it usually results in a series of smaller groups of related routines, making it easier to find a routine when you have a general idea of what you're looking for (a unit and a category, such as cursor manipulation), but you don't know if such a routine exists, or you know that it does but can't remember the name. The header at the top of the page shows the name of the unit in which the entries at the top of the page appear. The main purpose of the header is to make it easier to find the entries for a particular unit.

OpCrt Procedures and Functions

```
(1) 3-27       function MakeHiddenAttr(A : Byte) : Byte;
                 {-Make foreground and background colors the same}
(1) 3-23       function GetCrtMode : Byte;
                 {-Get the current video mode. Also reinitializes internal
                   variables. May reset: CurrentMode, ScreenWidth, ScreenHeight,
                   CurrentPage, and VideoSegment.}
(1) 3-24       procedure GotoXYAbs(X, Y : Byte);
                 {-Move cursor to column X, row Y. No error checking done.}
(1) 3-42       function WhereXY : Word;
                 {-Return absolute row and column coordinates of cursor. High byte
                   has current row (Y), low byte has current column (X).}
(1) 3-43       function WhereYAbs : Byte;
                 {-Return absolute row coordinate of cursor}
(1) 3-42       function WhereXAbs : Byte;
                 {-Return absolute column coordinate of cursor}
(1) 3-43       procedure WhereXYdirect(var X, Y : Byte);
                 {-Read the current position of the cursor directly from the CRT
                   controller}
(1) 3-39       procedure SetVisiblePage(PageNum : Byte);
                 {-Set current video page}
(1) 3-36       procedure SetActivePage(PageNo : Byte);
                 {-Selects the video page that will be written to with subsequent
                   operations on the screen. Does not affect Write/Ln or Read/Ln.}
(1) 3-24       function GetPageSegment(PageNo : Byte) : Word;
                 {-Get the video segment corresponding to the specified video page}
```

OPCRT

(1) 3-24 `function GetPagePointer(PageNo : Byte) : Pointer;`
`{-Get a pointer to the start of the specified video page}`

(1) 3-14 `procedure ClearPage(PageNo : Byte);`
`{-Clear the specified video page with TextChar and TextAttr}`

(1) 3-34 `procedure ScrollWindowUp(XLo, YLo, XHi, YHi, Lines : Byte);`
`{-Scrolls the designated window up the specified number of lines.}`

(1) 3-34 `procedure ScrollWindowDown(XLo, YLo, XHi, YHi, Lines : Byte);`
`{-Scrolls the designated window down the specified number of lines.}`

(1) 3-15 `function CursorTypeSL : Word;`
`{-Returns a word. High byte has starting scan line, low byte has ending.}`

(1) 3-15 `function CursorStartLine : Byte;`
`{-Returns the starting scan line of the cursor}`

(1) 3-15 `function CursorEndLine : Byte;`
`{-Returns the ending scan line of the cursor.}`

(1) 3-37 `procedure SetCursorSize(Startline, EndLine : Byte);`
`{-Sets the cursor's starting and ending scan lines.}`

(1) 3-29 `procedure NormalCursor;`
`{-Set normal scan lines for cursor based on current video mode}`

(1) 3-20 `procedure FatCursor;`
`{-Set larger scan lines for cursor based on current video mode}`

(1) 3-12 `procedure BlockCursor;`
`{-Set scan lines for a block cursor}`

(1) 3-26 `procedure HiddenCursor;`
`{-Hide the cursor}`

(1) 3-38 `procedure SetCursorType(CT : CursorType);`
`{-Sets the cursor shape to the specified type}`

(1) 3-13 `function ClassifyCursorType : CursorType;`
`{-Classify cursor type based on scan lines}`

(1) 3-31 `function ReadCharAtCursor : Char;`
`{-Returns character at the current cursor location on the selected page.}`

(1) 3-30 `function ReadAttrAtCursor : Byte;`
`{-Returns attribute at the current cursor location on the selected page.}`

(1) 3-23 `procedure GetCursorState(var XY, ScanLines : Word);`
`{-Return the current position and size of the cursor}`

(1) 3-32 `procedure RestoreCursorState(XY, ScanLines : Word);`
`{-Reset the cursor to a position and size saved with GetCursorState}`

(1) 3-19 `procedure FastWrite(St : string; Row, Col : Word; Attr : Byte);`
`{-Write St at Row,Col in Attr (video attribute) without snow}`

(1) 3-18 `procedure FastText(St : string; Row, Col : Word);`
`{-Write St at Row,Col without changing the underlying video attribute.}`

(1) 3-18 `procedure FastVert(St : string; Row, Col : Word; Attr : Byte);`
`{-Write St vertically at Row,Col in Attr (video attribute)}`

(1) 3-17 `procedure FastFill(Number : Word; Ch : Char; Row, Col : Word; Attr : Byte);`
`{-Fill Number chs at Row,Col in Attr (video attribute) without snow}`

(1) 3-16 `procedure FastCenter(St : string; Row, Attr : Byte);`
`{-Write St centered on window Row in Attr (video attribute) without snow}`

(1) 3-17 `procedure FastFlush(St : string; Row, Attr : Byte);`
`{-Write St flush right on window Row in Attr (video attribute) without snow}`

(1) 3-17 `procedure FastRead(Number : Byte; Row, Col : Word; var St : string);`
`{-Read Number characters from the screen into St starting at Row, Col}`

(1) 3-30 `procedure ReadAttribute(Number : Byte; Row, Col : Word; var St : string);`
`{-Read Number attributes from the screen into St starting at Row, Col}`

```
(1) 3-44    procedure WriteAttribute(St : String; Row, Col : Word);
              {-Write string of attributes St at Row,Col without changing
                characters}
(1) 3-12    procedure ChangeAttribute(Number : Word; Row, Col : Word;
                                      Attr : Byte);
              {-Change Number video attributes to Attr starting at Row,Col}
(1) 3-29    procedure MoveScreen(var Source, Dest; Length : Word);
              {-Move Length words from Source to Dest without snow}
(1) 3-19    procedure FastWriteAttr(St : String; Row, Col : Word;
                                    AttrSt : String);
              {-Write St at Row,Col using attributes in AttrSt}
(1) 3-21    procedure FlexWrite(St : string; Row, Col : Word;
                                var FAttrs : FlexAttrs);
              {-Write St at Row,Col with flexible color handling}
(1) 3-21    function FlexLen(S : string) : Byte;
              {-Return actual (visible) length of a flex string}
(1) 3-20    procedure FastWriteCtrl(St : String; Row, Col : Word;
                                    Attr, Ctrl : Byte);
              {-Write St at Row,Col in Attr (video attribute) without snow.
                Control characters displayed in Ctrl as upper-case letters}
(1) 3-33    function SaveWindow(XLow, YLow, XHigh, YHigh : Byte;
                                Allocate : Boolean;
                                var Covers : Pointer) : Boolean;
              {-Allocate buffer space if requested and save window contents}
(1) 3-32    procedure RestoreWindow(XLow, YLow, XHigh, YHigh : Byte;
                                    Deallocate : Boolean; var Covers : Pointer);
              {-Restore screen contents and deallocate buffer space if
                requested}
(1) 3-14    procedure ClearWindow(XLow, YLow, XHigh, YHigh : Word; Ch : Char;
                                  A : Byte);
              {-Clear a window using the specified character and attribute}
(1) 3-22    procedure FrameWindow(LeftCol, TopRow, RightCol, BotRow : Word;
                                  FAttr, HAttr : Byte; Header : string);
              {-Draws a frame around a window}
(1) 3-39    procedure StoreWindowCoordinates(var WC : WindowCoordinates);
              {-Store the window coordinates for the active window}
(1) 3-33    procedure RestoreWindowCoordinates(WC : WindowCoordinates);
              {-Restore previously saved window coordinates}
(1) 3-28    function MapColor(C : Byte) : Byte;
              {-Map a video attribute for visibility on mono/bw displays}
(1) 3-41    function UseColor : Boolean;
              {-Returns True if color video attributes should be used}
(1) 3-14    function ColorMono(Color, Mono : Byte) : Byte;
              {-Choose between a color and a monochrome video attribute}
(1) 3-28    function MapMono(Color, Mono : Byte) : Byte;
              {-Return a mapped color video attribute if Mono = $FF}
(1) 3-36    procedure SetBlink(On : Boolean);
              {-Enable text mode attribute blinking if On is True}
(1) 3-37    procedure SetCrtBorder(Attr : Byte);
              {-Set border to background color if card type and mode allow}
(1) 3-22    function Font8x8Selected : Boolean;
              {-Return True if EGA or VGA is active and in 8x8 font}
(1) 3-35    procedure SelectFont8x8(On : Boolean);
              {-Toggle 8x8 font on or off}
(1) 3-26    function HercPresent : Boolean;
              {-Return true if a Hercules graphics card is present}
(1) 3-41    procedure SwitchInColorCard(ColorOn : Boolean);
              {-Activate or deactivate colors on a Hercules InColor card}
(1) 3-25    function HercGraphicsMode : Boolean;
              {-Return True if a Hercules card is in graphics mode}
(1) 3-25    function HercModeTestWorks : Boolean;
              {-Return True if HercGraphicsMode will work}
(1) 3-38    procedure SetHercMode(GraphMode : Boolean; GraphPage : Byte);
              {-Set Hercules card to graphics mode or text mode, and activate
                specified graphics page (if switching to graphics mode).}
```

```
(1) 3-31    function ReadKeyWord : Word;
              {-Waits for keypress, then returns scan and character codes
                together}
(1) 3-13    function CheckKbd(var KeyCode : Word) : Boolean;
              {-Returns True (and the key codes) if a keystroke is waiting}
(1) 3-27    function KbdFlags : Byte;
              {-Returns keyboard status flags as a bit-coded byte}
(1) 3-16    function EnhancedKbdInstalled : Boolean;
              {-Returns True if an enhanced keyboard is installed}
(1) 3-40    procedure StuffKey(W : Word);
              {-Stuff one key into the keyboard buffer}
(1) 3-40    procedure StuffString(S : string);
              {-Stuff the contents of S into the keyboard buffer}
(1) 3-32    procedure ReInitCrt;
              {-Reinitialize CRT unit's internal variables}
```

OpCrt ColorSet

```
(1) 3-48    procedure SetTextAttr(Color, Mono : Byte);
              {-Set attributes for ordinary text}
(1) 3-48    procedure SetCtrlAttr(Color, Mono : Byte);
              {-Set attributes for control characters}
(1) 3-48    procedure SetFrameAttr(Color, Mono : Byte);
              {-Set attributes for window frames}
(1) 3-48    procedure SetHeaderAttr(Color, Mono : Byte);
              {-Set attributes for window headers}
(1) 3-48    procedure SetShadowAttr(Color, Mono : Byte);
              {-Set attributes for window shadows}
(1) 3-48    procedure SetHighlightAttr(Color, Mono : Byte);
              {-Set attributes used to highlight found text}
(1) 3-48    procedure SetPromptAttr(Color, Mono : Byte);
              {-Set attributes for entry field prompts}
(1) 3-48    procedure SetSelectedPromptAttr(Color, Mono : Byte);
              {-Set attributes for selected entry field prompts}
(1) 3-48    procedure SetProtectedPromptAttr(Color, Mono : Byte);
              {-Set attributes for protected entry field prompts}
(1) 3-48    procedure SetFieldAttr(Color, Mono : Byte);
              {-Set attributes for unselected entry fields}
(1) 3-48    procedure SetSelectedFieldAttr(Color, Mono : Byte);
              {-Set attributes for selected entry field fields}
(1) 3-48    procedure SetProtectedFieldAttr(Color, Mono : Byte);
              {-Set attributes for protected entry fields}
(1) 3-48    procedure SetScrollBarAttr(Color, Mono : Byte);
              {-Set attributes for scroll bars}
(1) 3-48    procedure SetSliderAttr(Color, Mono : Byte);
              {-Set attributes for sliders in scroll bars}
(1) 3-48    procedure SetHotSpotAttr(Color, Mono : Byte);
              {-Sets attributes for mouse hot spots}
(1) 3-48    procedure SetBlockAttr(Color, Mono : Byte);
              {-Set attributes for marked blocks}
(1) 3-48    procedure SetMarkerAttr(Color, Mono : Byte);
              {-Set attributes for text markers}
(1) 3-48    procedure SetDelimAttr(Color, Mono : Byte);
              {-Set attributes for unselected entry field delimiters}
(1) 3-48    procedure SetSelectedDelimAttr(Color, Mono : Byte);
              {-Set attributes for selected entry field delimiters}
(1) 3-48    procedure SetProtectedDelimAttr(Color, Mono : Byte);
              {-Set attributes for protected entry field delimiters}
(1) 3-48    procedure SetSelectedItemAttr(Color, Mono : Byte);
              {-Set attributes for selected pick and menu items}
(1) 3-48    procedure SetProtectedItemAttr(Color, Mono : Byte);
              {-Set attributes for protected pick and menu items}
(1) 3-48    procedure SetHighlightItemAttr(Color, Mono : Byte);
              {-Set attributes for highlighted selection characters for menu
                items}
```

```
(1) 3-48    procedure SetFlexAHelpAttr(Color, Mono : Byte);
              {-Set attributes for help text surrounded by ^A}
(1) 3-48    procedure SetFlexBHelpAttr(Color, Mono : Byte);
              {-Set attributes for help text surrounded by ^B}
(1) 3-48    procedure SetFlexCHelpAttr(Color, Mono : Byte);
              {-Set attributes for help text surrounded by ^C}
(1) 3-48    procedure SetUnselXrefAttr(Color, Mono : Byte);
              {-Set attributes for unselected cross-references in help text}
(1) 3-48    procedure SetSelXrefAttr(Color, Mono : Byte);
              {-Set attributes for selected cross-references in help text}
(1) 3-48    procedure SetMouseAttr(Color, Mono : Byte);
              {-Set attributes for mouse cursor}
```

OpMouse Procedures and Functions

```
(1) 3-71    function MousePressed : Boolean;
              {-Return True if a mouse button is currently being pressed}
(1) 3-71    function MouseKeyWord : Word;
              {-Return a pseudo-scan code based on which mouse button is being
                pressed}
(1) 3-68    function KeyOrButtonPressed : Boolean;
              {-Return True if a key or mouse button has been pressed}
(1) 3-76    function ReadKeyOrButton : Word;
              {-Return next key or mouse button}
(1) 3-64    procedure EnableEventHandling;
              {-Enable the event handler needed for MousePressed and
                MouseKeyWord}
(1) 3-63    procedure DisableEventHandling;
              {-Disable the event handler installed by EnableEventHandling}
(1) 3-68    procedure InitializeMouse;
              {-Reinitializes mouse and sets MouseInstalled}
(1) 3-80    procedure ShowMouse;
              {-Show the mouse cursor}
(1) 3-67    procedure HideMouse;
              {-Hide the mouse cursor}
(1) 3-73    procedure MouseWhereXY(var MouseX, MouseY : Byte;
                                   var Status : ButtonStatus);
              {-Return mouse position and button status}
(1) 3-72    function MouseWhereX : Byte;
              {-Return current X coordinate for mouse}
(1) 3-74    function MouseWhereY : Byte;
              {-Return current Y coordinate for mouse}
(1) 3-74    procedure MouseWhereXYabs(var MouseX, MouseY : Byte;
                                      var Status : ButtonStatus);
              {-Return mouse position (absolute) and button status}
(1) 3-73    function MouseWhereXabs : Byte;
              {-Return current X coordinate (absolute) for mouse}
(1) 3-75    function MouseWhereYabs : Byte;
              {-Return current Y coordinate (absolute) for mouse}
(1) 3-70    procedure MouseGotoXY(MouseX, MouseY : Byte);
              {-Set mouse position}
(1) 3-69    function MouseButtonPressed(Button : ButtonStatus; var Count : Word;
                                        var LastX, LastY : Byte) : Boolean;
              {-Returns True if the Button to check has been pressed. If so,
                Count has the number of times it has been pressed, and
                LastX/LastY have its position the last time it was pressed.}
(1) 3-69    function MouseButtonReleased(Button : ButtonStatus; var Count : Word;
                                         var LastX, LastY : Byte) : Boolean;
              {-Returns True if the Button to check has been released. If so,
                Count has the number of times it has been released, and
                LastX/LastY have its position the last time it was released.}
(1) 3-75    procedure MouseWindow(XLow, YLow, XHigh, YHigh : Byte);
              {-Sets window coordinates to be observed by the mouse}
(1) 3-65    procedure FullMouseWindow;
              {-Sets mouse window coordinates to full screen}
```

OPMOUSE

(1) 3-81
```
procedure StoreMouseCoordinates(var WC : WindowCoordinates);
  {-Store the mouse window coordinates in WC}
```
(1) 3-77
```
procedure RestoreMouseCoordinates(WC : WindowCoordinates);
  {-Restore previously saved mouse window coordinates}
```
(1) 3-70
```
function MouseInWindow(XLo, YLo, XHi, YHi : Byte) : Boolean;
  {-Return True if mouse is within the specified window}
```
(1) 3-81
```
procedure SoftMouseCursor(ScreenMask, CursorMask : Word);
  {-Set mouse to use a software cursor}
```
(1) 3-66
```
procedure HardMouseCursor(StartLine, EndLine : Word);
  {-Set mouse to use the hardware cursor. StartLine and EndLine
    specify the shape of the cursor.}
```
(1) 3-76
```
procedure NormalMouseCursor;
  {-Set normal scan lines for mouse cursor based on current video
    mode}
```
(1) 3-64
```
procedure FatMouseCursor;
  {-Set larger scan lines for mouse cursor based on current video
    mode}
```
(1) 3-63
```
procedure BlockMouseCursor;
  {-Set scan lines for a block mouse cursor}
```
(1) 3-67
```
procedure HiddenMouseCursor;
  {-Hide the mouse cursor}
```
(1) 3-65
```
procedure GetMickeyCount(var Horizontal, Vertical : Integer);
  {-Returns the horizontal and vertical mickey count since the last
    call to this function. Negative numbers indicate movement up or
    to the left; positive numbers indicate movement down or to the
    right.}
```
(1) 3-78
```
procedure SetMickeyToPixelRatio(Horizontal, Vertical : Integer);
  {-Sets the mickey-to-pixel ratio. Default setting is 8,16. A
    setting of 16,32 slows down the mouse considerably. A setting of
    4,8 makes the mouse fly.}
```
(1) 3-79
```
procedure SetMouseEventHandler(EventMask : MouseEventType;
                               UserRoutine : Pointer);
  {-Sets the address of a routine to be called when the specified
    mouse events occur}
```
(1) 3-66
```
function GetMousePage : Byte;
  {-Returns the video page where the mouse is being displayed}
```
(1) 3-79
```
procedure SetMousePage(Page : Byte);
  {-Sets the video page where the mouse will be displayed}
```
(1) 3-72
```
function MouseStateBufferSize : Word;
  {-Returns amount of memory needed to save the state of the mouse
    driver}
```
(1) 3-78
```
procedure SaveMouseState(var MSP : MouseStatePtr;
                         Allocate : Boolean);
  {-Save the state of the mouse driver, allocating the buffer if
    requested.}
```
(1) 3-77
```
procedure RestoreMouseState(var MSP : MouseStatePtr;
                            Deallocate : Boolean);
  {-Restore the state of the mouse driver and Deallocate the buffer
    if requested}
```
(1) 3-67
```
procedure HideMousePrim(var MouseState : Boolean);
  {-Save state of mouse cursor in MouseState and hide it}
```
(1) 3-80
```
procedure ShowMousePrim(MouseState : Boolean);
  {-Hide or unhide the mouse cursor}
```

OpCmd CommandProcessor

(1) 3-105
```
constructor Init(KeysPtr : CmdTablePtr; MaxIndex : Word);
  {-Initialize a command processor with only one command table}
```
(1) 3-105
```
constructor InitCustom(PrimaryKeys : CmdTablePtr;
                       PrimaryMaxIndex : Word;
                       SecondaryKeys : CmdTablePtr;
                       SecondaryMaxIndex : Word; Options : Word);
  {-Initialize a command processor with custom options}
```
(1) 3-100
```
destructor Done; virtual;
  {-Dispose of any memory that was allocated by the command
    processor}
```

```
(1) 3-102   function GetCommand(var Key : Word) : Word;
              {-Get next command or character}
(1) 3-95    procedure AddCommand(Cmd : Word; NumKeys : Byte; Key1, Key2 : Word);
              {-Add a new primary command key assignment or change an existing
                one}
(1) 3-96    procedure AddSecondaryCommand(Cmd : Word; NumKeys : Byte;
                                          Key1, Key2 : Word);
              {-Add a new secondary command key assignment or change an existing
                one}
(1) 3-104   function GetLastError : Word;
              {-Returns last error code and clears it}
(1) 3-108   procedure SetGetKeyProc(GKP : GetKeyProc);
              {-Change the cpGetKey routine}
(1) 3-109   procedure SetKeyPressedProc(KPP : KeyPressedProc);
              {-Change the cpKeyPressed routine}
(1) 3-108   procedure SetHelpProc(GHP : GetHelpProc);
              {-Change the HelpProc field}
(1) 3-110   procedure SetUserHookProc(UHP : UserHookProc);
              {-Specify a routine to be called each time a keystroke is
                evaluated}
(1) 3-99    procedure cpOptionsOn(OptionFlags : Word);
              {-Activate multiple options}
(1) 3-99    procedure cpOptionsOff(OptionFlags : Word);
              {-Deactivate multiple options}
(1) 3-99    function cpOptionsAreOn(OptionFlags : Word) : Boolean;
              {-Return true if all specified options are on}
(1) 3-98    function cpGetKey : Word; virtual;
              {-Called to get next keystroke}
(1) 3-98    function cpKeyPressed : Boolean; virtual;
              {-Called to determine if a key has been pressed}
(1) 3-98    procedure cpGetHelp(UnitCode : Byte; IdPtr : Pointer;
                                HelpIndex : Word); virtual;
              {-Called when help is requested}
(1) 3-100   procedure cpUserHook(MT : MatchType; Key : Word); virtual;
              {-Called each time a keystroke is processed}
(1) 3-109   procedure SetKeyPtr(KeysPtr : CmdTablePtr; MaxIndex : Word);
              {-Change the primary command table}
(1) 3-103   function GetKeyPtr(var MaxIndex : Word) : CmdTablePtr;
              {-Return a pointer to the primary command table}
(1) 3-110   procedure SetSecondaryKeyPtr(KeysPtr : CmdTablePtr; MaxIndex : Word);
              {-Change the secondary command table}
(1) 3-104   function GetSecondaryKeyPtr(var MaxIndex : Word) : CmdTablePtr;
              {-Return a pointer to the secondary command table}
(1) 3-103   procedure GetKeyAssignment(Cmd : Word; var NumKeys : Byte;
                                       var Key1, Key2 : Word);
              {-Return one set of key(s) assigned to the specified command.
                NumKeys = 0 if none found.}
(1) 3-101   function GetCmdAssignment(NumKeys : Byte; Key1, Key2 : Word;
                                      var Primary : Boolean) : Word;
              {-Return command associated with specified key combination, or
                ccNone if none found. Primary is False only if match found in
                secondary key set.}
(1) 3-96    function CheckForConflict(NumKeys : Byte;
                                      Key1, Key2 : Word) : MatchType;
              {-Check to see if the specified key combination conflicts with an
                existing one}
(1) 3-107   procedure SetCommandList(CmdList : Pointer; NumCmds : Word);
              {-Specify a list of commands to be auto-executed}
(1) 3-97    function CommandStringPending : Boolean;
              {-Returns True if a text string is pending in the list of
                auto-executed commands}
(1) 3-102   function GetCommandString : string;
              {-Returns a text string when CommandStringPending is True}
(1) 3-106   constructor Load(var S : IdStream);
              {-Load a command processor from a stream}
```

OPCMD

(1) 3-111
```
procedure Store(var S : IdStream);
  {-Store a command processor in a stream}
```
(1) 3-107
```
function MouseEnabled : Boolean;
  {-Return True if mouse support enabled}
```

OpCmd CommandPacker

(1) 3-116
```
constructor Init(PKeys : CmdTablePtr; MaxCmd, MaxIndex : Word;
                 UKeys : UnpackedCmdPtr; Columns : Byte;
                 CmdsAreWords : Boolean);
  {-Initialize pointers to packed/unpacked command tables}
```
(1) 3-116
```
function PackKeys : Boolean;
  {-Convert unpacked array into a packed list of keys again. Returns
    False if keys won't all fit.}
```
(1) 3-116
```
function SizeKeys : Word;
  {-Return number of bytes in packed version of UnpackedKeys}
```
(1) 3-115
```
function GetKeyRecPtr(Cmd : Word; Col : Byte) : KeyRecPtr;
  {-Returns a pointer to the KeyRec for the Col'th instance of Cmd}
```
(1) 3-115
```
function ConflictsFound : Boolean;
  {-Check unpacked commands for conflicts. Returns True if conflicts
    found}
```

OpCmd Procedures and Functions

(1) 3-118
```
procedure SetBell(Pitch, Duration : Word);
  {-Set pitch and duration for bell}
```
(1) 3-118
```
procedure RingBell;
  {-Ring the bell}
```
(1) 3-118
```
procedure DefaultErrorProc(UnitCode : Byte; var ErrorCode : Word;
                           ErrorMsg : string);
  {-Default error handler--just beeps}
```

OpFrame AbstractFrame

(1) 4-42
```
procedure ClearErrors;
  {-Clear all pending errors}
```

OpFrame ScreenRect

(1) 4-11
```
constructor Alloc(X1, Y1, X2, Y2 : Byte);
  {-Initialize coordinates and allocate a screen buffer}
```
(1) 4-15
```
constructor Init(X1, Y1, X2, Y2 : Byte; BufPtr : Pointer);
  {-Initialize coordinates and assign a preallocated buffer}
```
(1) 4-14
```
destructor Done; virtual;
  {-Deallocate buffer}
```
(1) 4-12
```
function BufSize : Word;
  {-Return required size of buffer in bytes}
```
(1) 4-12
```
procedure Clear(FChar : Char; FAttr : Byte);
  {-Clear with character and attribute}
```
(1) 4-13
```
procedure CopyFromScreen;
  {-Copy from screen to buffer}
```
(1) 4-14
```
procedure CopyToScreen;
  {-Copy buffer to screen}
```
(1) 4-13
```
procedure CopyPartToScreen(X1, Y1, X2, Y2 : Byte);
  {-Copy part of the buffer to the screen}
```
(1) 4-10
```
procedure Adjust(X1, Y1, X2, Y2 : Byte);
  {-Adjust buffer coordinates to X1, Y1, X2, Y2}
```
(1) 4-19
```
procedure Transfer(X1, Y1, X2, Y2 : Byte;
                   FChar : Char; FAttr : Byte);
  {-Transfer buffer to new buffer at coordinates X1, Y1, X2, Y2}
```
(1) 4-17
```
function srResult : Word;
  {-Return most recent status code}
```
(1) 4-19
```
procedure StoreContents(On : Boolean);
  {-Specify whether Store writes contents of screen buffer}
```

```
(1) 4-15    function HaveContents : Boolean;
              {-Return True if contents were loaded by Load}
(1) 4-12    procedure Coordinates(var X1, Y1, X2, Y2 : Byte);
              {-Return coordinates of screen rectangle}
(1) 4-16    constructor Load(var S : IdStream);
              {-Load from stream}
(1) 4-18    procedure Store(var S : IdStream);
              {-Store to stream}
```

OpFrame Frame

```
(1) 4-48    constructor Init(X1, Y1, X2, Y2 : Byte; FA : FrameArray;
                             var Colors : ColorSet);
              {-Initialize frame coordinates and all fields}
(1) 4-46    constructor fCopy(var F : Frame);
              {-Initialize a frame from another, making a unique copy of all
                dynamic structures}
(1) 4-44    destructor Done; virtual;
              {-Destroy a frame}
(1) 4-50    procedure SetFrameType(FA : FrameArray);
              {-Change the frame type}
(1) 4-50    procedure SetFrameAttr(Color, Mono : Byte); virtual;
              {-Set attributes for frame characters}
(1) 4-51    procedure SetHeaderAttr(Color, Mono : Byte; ChangeAll : Boolean);
              {-Set attributes for header characters}
(1) 4-51    procedure SetShadowAttr(Color, Mono : Byte; ChangeAll : Boolean);
              {-Set attributes for shadow characters}
(1) 4-34    procedure AddHeader(S : string; Posn : HeaderPosType);
              {-Add a standard header}
(1) 4-34    procedure AddHeaderColor(S : string; Posn : HeaderPosType;
                                     AttrColor, AttrMono : Byte);
              {-Add a standard header with custom color}
(1) 4-31    procedure AddCustomHeader(S : string; Posn : FrameCornerType;
                                      DX, DY : Integer;
                                      AttrColor, AttrMono : Byte);
              {-Add a custom header}
(1) 4-39    procedure AddSpanHeader(FirstChar, SpanChar, LastChar : Char;
                                    Dpos : Integer; Posn : FrameEdgeType);
              {-Add a header spanning the frame}
(1) 4-39    procedure AddSpanHeaderColor(FirstChar, SpanChar, LastChar : Char;
                                         Dpos : Integer; Posn : FrameEdgeType;
                                         AttrColor, AttrMono : Byte);
              {-Add a header spanning the frame with custom color}
(1) 4-47    function GetLastHeaderIndex : Byte;
              {-Return index of last header added}
(1) 4-41    procedure ChangeHeaderString(Index : Byte; S : string;
                                         var Redraw : Boolean);
              {-Change existing header string and update position}
(1) 4-40    procedure ChangeHeaderAttr(Index : Byte; Color, Mono : Byte);
              {-Change existing header colors}
(1) 4-45    procedure DrawHeader(Index : Byte);
              {-Draw specified header}
(1) 4-44    procedure DisableHeader(Index : Byte; On : Boolean);
              {-Disable or enable specified header}
(1) 4-37    procedure AddShadow(Posn : ShadowPosType; DrawType : ShadowDrawType);
              {-Add a standard shadow}
(1) 4-38    procedure AddShadowColor(Posn : ShadowPosType;
                                     DrawType : ShadowDrawType;
                                     AttrColor, AttrMono : Byte);
              {-Add a standard shadow with custom color}
(1) 4-33    procedure AddCustomShadow(ShChar : Char; Edge : FrameEdgeType;
                                      DX, DY : Integer; W, H : Byte;
                                      AttrColor, AttrMono : Byte);
              {-Add a custom shadow rectangle}
```

OPFRAME

```
(1) 4-37    procedure AddScrollBar(Posn : FrameEdgeType;
                                   MinUser, MaxUser : LongInt;
                                   var Colors : ColorSet);
              {-Add a standard scroll bar}
(1) 4-33    procedure AddCustomScrollBar(Posn : FrameEdgeType;
                                         MinUser, MaxUser : LongInt;
                                         DecOffset, IncOffset : Byte;
                                         SliChar, BarChar : Char;
                                         var Colors : ColorSet);
              {-Add a custom scroll bar}
(1) 4-36    procedure AddHotSpot(Posn : FrameCornerType; HotChar : Char;
                                 var Colors : ColorSet);
              {-Add a hot spot in a frame corner}
(1) 4-35    procedure AddHotBar(Posn : FrameEdgeType; Code : Byte);
              {-Add a hot region covering an entire frame edge}
(1) 4-36    procedure AddHotRegion(Posn : FrameCharType; Code : Byte;
                                   DX, DY : Integer; Width, Height : Byte);
              {-Add a fixed size hot region relative to the frame corner}
(1) 4-35    procedure AddHotHeader(hType : HeaderPosType; Code : Byte;
                                   Len : Byte);
              {-Add a hot region to coincide with a header}
(1) 4-48    function GetLastHotIndex : Byte;
              {-Return index of last hot region}
(1) 4-42    procedure ChangeHotHeader(Index : Byte; Len : Byte);
              {-Change the length (and position) of a hot header}
(1) 4-52    procedure SetSizeLimits(MinW, MinH, MaxW, MaxH : Byte);
              {-Set limits for sizing of frame}
(1) 4-50    procedure SetClipLimits(MinXL, MinYL, MaxXH, MaxYH : Byte);
              {-Set limits for drawing frame (shadows get clipped)}
(1) 4-52    procedure SetSliderValue(Posn : FrameEdgeType; UserVal : LongInt);
              {-Change the user value without drawing it}
(1) 4-43    function CurUserValue(Posn : FrameEdgeType) : LongInt;
              {-Return the current user value for specified scroll bar}
(1) 4-45    procedure Draw; virtual;
              {-Draw the frame, scrollbars, headers, and shadows}
(1) 4-53    procedure UpdateFrame; virtual;
              {-Draw the frame, scrollbars, and headers (no shadows)}
(1) 4-49    function IsFramed : Boolean;
              {-Return True if any of the FrameChars are not NoFrameChar}
(1) 4-43    procedure Coordinates(var X1, Y1, X2, Y2 : Byte);
              {-Return the coordinates of the frame}
(1) 4-54    procedure WithinFrameCoords(var X1, Y1, X2, Y2 : Byte);
              {-Return the coordinates of region within, not including, the
                border}
(1) 4-40    procedure AdjustFrame(X1, Y1, X2, Y2 : Byte);
              {-Set new coordinates and adjust related structures}
(1) 4-46    procedure Error(Code : Word); virtual;
              {-Report that an error occurred}
(1) 4-47    function GetLastError : Word; virtual;
              {-Return and clear the last error code, 0 if none}
(1) 4-49    constructor Load(var S : IdStream);
              {-Load from stream}
(1) 4-53    procedure Store(var S : IdStream);
              {-Store to stream}
```

OpWindow AbstractWindow

```
(1) 4-107   function Width : Word;
              {-Return the active width of the window}
(1) 4-84    function Height : Word;
              {-Return the active height of the window}
(1) 4-69    procedure ClearErrors;
              {-Clear all pending errors}
```

(1) 4-85
```
constructor Init(X1, Y1, X2, Y2 : Byte);
  {-Create a window with default options and colors}
```
(1) 4-85
```
constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                       Options : LongInt);
  {-Create a window with custom options}
```
(1) 4-106
```
constructor wCopy(var W : RawWindow);
  {-Initialize a window from another one, making a unique copy of
    all dynamic structures}
```
(1) 4-71
```
destructor Done; virtual;
  {-Destroy a window}
```
(1) 4-63
```
procedure AdjustFrameCoords(X1, Y1, X2, Y2 : Byte);
  {-Change frame coordinates of an inactive window.
    Completely clears existing frame definition}
```
(1) 4-71
```
procedure DeltaAdjustFrame(dX1, dY1, dX2, dY2 : Integer);
  {-Change frame coordinates of an inactive window.
    Reallocates wCovers buffer but changes nothing else}
```
(1) 4-97
```
procedure SetSizeLimits(MinW, MinH, MaxW, MaxH : Byte);
  {-Set limits for sizing of active window}
```
(1) 4-96
```
procedure SetPosLimits(MinX, MinY, MaxX, MaxY : Byte);
  {-Set limits for moving and drawing (generally limited by frame)}
```
(1) 4-96
```
procedure SetFrameLimits(MinW, MinH, MaxW, MaxH : Byte);
  {-Set limits for sizing of frame}
```
(1) 4-103
```
procedure UpdateScreenSize;
  {-Update window size, position, and limits for current screen
    size}
```
(1) 4-77
```
procedure EnableExplosions(StageDelay : Word);
  {-Enable exploding windows with StageDelay milliseconds per stage}
```
(1) 4-75
```
procedure EnableAlternateOpen(OpenProc, CloseProc : RawWindowProc;
                              StageDelay : Word);
  {-Enable alternate draw and erase routines}
```
(1) 4-77
```
procedure EnableNormalOpen;
  {-Disable exploding windows or alternate draw routines}
```
(1) 4-109
```
procedure wOptionsOn(OptionCodes : LongInt);
  {-Turn options on}
```
(1) 4-108
```
procedure wOptionsOff(OptionCodes : LongInt);
  {-Turn options off}
```
(1) 4-107
```
function wOptionsAreOn(OptionCodes : LongInt) : Boolean;
  {-Return true if all specified options are on}
```
(1) 4-97
```
procedure SetTextAttr(Color, Mono : Byte); virtual;
  {-Set attributes for normal window text}
```
(1) 4-95
```
procedure SetBackChar(BackChar : Char); virtual;
  {-Set background character}
```
(1) 4-106
```
procedure wGotoXY(X, Y : Integer); virtual;
  {-Move cursor to window-relative coordinates}
```
(1) 4-110
```
procedure wWhereXY(var X, Y : Integer); virtual;
  {-Return window-relative coordinates of cursor}
```
(1) 4-110
```
procedure wWhereXYAbs(var X, Y : Byte);
  {-Return absolute coordinates of cursor}
```
(1) 4-95
```
procedure SetCursor(CT : CursorType);
  {-Set cursor shape}
```
(1) 4-83
```
function GetCursor : CursorType;
  {-Return current cursor shape}
```
(1) 4-67
```
procedure ChangeHeader(Index : Byte; S : string);
  {-Change header string, and if window is current, update screen}
```
(1) 4-86
```
function InWindow(AX, AY : Byte) : Boolean;
  {-Return true if specified absolute position is within active
    window}
```
(1) 4-70
```
function CursorInWindow : Boolean;
  {-Return true if cursor is currently within active window}
```
(1) 4-80
```
procedure EvaluatePos(X, Y : Byte);
  {-Evaluate specified absolute position relative to window,
    hotspots, and scrollbars. Access returned values using
    PosResults}
```

OPWINDOW

(1) 4-79 procedure EvaluateMousePos;
{-Evaluate current mouse position as with EvaluatePos}

(1) 4-91 function PosResults(var FramePos : FramePosType;
var HotCode : Byte) : LongInt;
{-Return values from most recent call to EvaluatePos or EvaluateMousePos}

(1) 4-79 procedure Error(Code : Word); virtual;
{-Report that an error occurred}

(1) 4-84 function GetLastError : Word; virtual;
{-Return and clear the last error code, 0 if none}

(1) 4-86 function IsActive : Boolean; virtual;
{-Return True if window is on-screen}

(1) 4-86 function IsCurrent : Boolean; virtual;
{-Return True if window is the current one}

(1) 4-87 function IsZoomed : Boolean; virtual;
{-Return True if window is currently zoomed}

(1) 4-89 function MainFramePtr : AbstractFramePtr; virtual;
{-Return a pointer to frame used when window is current}

(1) 4-65 function AltFramePtr : AbstractFramePtr; virtual;
{-Return a pointer to frame used when window is active but not current}

(1) 4-62 function ActiveFramePtr : AbstractFramePtr; virtual;
{-Return a pointer to frame the window is currently displaying, nil if none}

(1) 4-70 procedure Coordinates(var X1, Y1, X2, Y2 : Byte);
{-Return the active coordinates of the window}

(1) 4-81 procedure Extent(var X1, Y1, X2, Y2 : Byte;
IncludeShadows : Boolean);
{-Return the complete extent of the window, counting shadows if requested}

(1) 4-72 procedure Draw; virtual;
{-Draw or update a window}

(1) 4-102 procedure UpdateContents; virtual;
{-Update the contents of the currently displayed window}

(1) 4-78 procedure Erase; virtual;
{-Erase a window}

(1) 4-93 procedure SaveContents; virtual;
{-Save current window contents in the contents buffer}

(1) 4-74 procedure DrawSlider(Posn : FrameEdgeType; UserVal : LongInt);
{-Draw or update the slider}

(1) 4-78 procedure EraseSlider(Posn : FrameEdgeType);
{-Erase the slider}

(1) 4-100 function TweakSlider(Posn : FrameEdgeType; MousePosn : Byte;
UserVal : LongInt; Step : LongInt) : LongInt;
{-Adjust user value until slider exactly matches mouse position}

(1) 4-68 procedure ChangeScrollBar(Posn : FrameEdgeType;
MinUser, MaxUser : LongInt);
{-Change user range of existing scroll bar}

(1) 4-66 procedure ChangeAllScrollBars(MinHoriz, MaxHoriz : LongInt;
MinVert, MaxVert : LongInt); virtual;
{-Change user ranges of existing scroll bars. Erases sliders}

(1) 4-73 procedure DrawAllSliders(HorizVal, VertVal : LongInt); virtual;
{-Draw or update all sliders}

(1) 4-68 procedure Clear; virtual;
{-Clear active window area with wTextXXXX and wBackChar}

(1) 4-69 procedure ClearContents;
{-Clear region within frame with wTextXXXX and wBackChar}

(1) 4-90 procedure MoveWindow(DX, DY : Integer); virtual;
{-Move a window}

(1) 4-92 procedure ResizeWindow(DX, DY : Integer); virtual;
{-Resize a window}

(1) 4-64 procedure AdjustWindow(X1, Y1, X2, Y2 : Word); virtual;
{-Set new coordinates and adjust all related structures}

(1) 4-111 procedure Zoom; virtual;
{-Increase window size to maximum limits temporarily}

```
(1) 4-101     procedure Unzoom; virtual;
                {-Return zoomed window to pre-zoomed size}
(1) 4-94      procedure ScrollHoriz(Cols : Integer); virtual;
                {-Scroll contents horizontally}
(1) 4-94      procedure ScrollVert(Rows : Integer); virtual;
                {-Scroll contents vertically}
(1) 4-89      procedure MarkCurrent; virtual;
                {-Called when window is made current}
(1) 4-89      procedure MarkNotCurrent; virtual;
                {-Called when window is deselected}
(1) 4-104     procedure wFastWrite(St : string; Row, Col, Attr : Byte);
                {-Write a string}
(1) 4-104     procedure wFastText(St : string; Row, Col : Byte);
                {-Write St at Row,Col without changing the underlying video
                  attribute}
(1) 4-104     procedure wFastVert(St : string; Row, Col, Attr : Byte);
                {-Write a string vertically}
(1) 4-104     procedure wFastFill(Number : Word; Ch : Char; Row, Col, Attr : Byte);
                {-Fill Number chars at Row,Col in Attr}
(1) 4-104     procedure wChangeAttribute(Number : Word; Row, Col, Attr : Byte);
                {-Change Number video attributes to Attr starting at Row,Col}
(1) 4-104     procedure wFastCenter(St : string; Row, Attr : Byte);
                {-Write St centered on Row in Attr}
(1) 4-104     procedure wFastFlush(St : string; Row, Attr : Byte);
                {-Write St flush right on Row in Attr}
(1) 4-104     procedure wFastRead(Number, Row, Col : Byte; var St : string);
                {-Read Number characters from the screen into St starting at
                  Row, Col}
(1) 4-104     procedure wReadAttribute(Number, Row, Col : Byte; var St : string);
                {-Read Number attributes from the screen into St starting at
                  Row, Col}
(1) 4-104     procedure wWriteAttribute(St : String; Row, Col : Byte);
                {-Write string of attributes St at Row,Col without changing
                  characters}
(1) 4-104     procedure wFlexWrite(St : string; Row, Col : Byte;
                                   var FAttrs : FlexAttrs);
                {-Write a string flexibly using window-relative coordinates}
(1) 4-104     procedure wFastWriteCtrl(St : String; Row, Col, Attr, Ctrl : Byte);
                {-Write St at Row,Col in Attr (video attribute) without snow.
                  Control characters displayed in Ctrl as upper-case letters}
(1) 4-82      procedure fFastWrite(St : string; Row, Col, Attr : Byte);
                {-Write a string}
(1) 4-82      procedure fFastText(St : string; Row, Col : Byte);
                {-Write St at Row,Col without changing the underlying video
                  attribute}
(1) 4-82      procedure fFastVert(St : string; Row, Col, Attr : Byte);
                {-Write a string vertically}
(1) 4-82      procedure fFastFill(Number : Word; Ch : Char; Row, Col, Attr : Byte);
                {-Fill Number chars at Row,Col in Attr}
(1) 4-82      procedure fChangeAttribute(Number : Word; Row, Col, Attr : Byte);
                {-Change Number video attributes to Attr starting at Row,Col}
(1) 4-82      procedure fFastCenter(St : string; Row, Attr : Byte);
                {-Write St centered on Row in Attr}
(1) 4-82      procedure fFastFlush(St : string; Row, Attr : Byte);
                {-Write St flush right on Row in Attr}
(1) 4-82      procedure fFastRead(Number, Row, Col : Byte; var St : string);
                {-Read Number characters from the screen into St starting at
                  Row, Col}
(1) 4-82      procedure fReadAttribute(Number, Row, Col : Byte; var St : string);
                {-Read Number attributes from the screen into St starting at
                  Row, Col}
(1) 4-82      procedure fWriteAttribute(St : String; Row, Col : Byte);
                {-Write string of attributes St at Row,Col without changing
                  characters}
```

```
(1) 4-82    procedure fFlexWrite(St : string; Row, Col : Byte;
                                 var FAttrs : FlexAttrs);
              {-Write a string flexibly using window-relative coordinates}
(1) 4-82    procedure fFastWriteCtrl(St : String; Row, Col, Attr, Ctrl : Byte);
              {-Write St at Row,Col in Attr (video attribute) without snow.
                Control characters displayed in Ctrl as upper-case letters}
(1) 4-88    constructor Load(var S : IdStream);
              {-Load from stream}
(1) 4-98    procedure Store(var S : IdStream);
              {-Store to stream}
```

OpWindow StackWindow

```
(1) 4-122   constructor Init(X1, Y1, X2, Y2 : Byte);
              {-Create a stacked window with default options and colors}
(1) 4-122   constructor InitCustom(X1, Y1, X2, Y2 : Byte;
                                   var Colors : ColorSet; Options : LongInt);
              {-Create a window with custom options}
(1) 4-129   constructor wCopy(var S : StackWindow);
              {-Initialize a window from another one, making a unique copy of
                all dynamic structures}
(1) 4-126   procedure SetStackProc(SP : StackWindowProc);
              {-Set window stack proc}
(1) 4-129   function WindowNumber : Word;
              {-Return window number. 0 if inactive, else 1 or greater}
(1) 4-119   procedure AssignNumber; virtual;
              {-Called to assign window number}
(1) 4-120   procedure Draw; virtual;
              {-Draw or update a window and push it on stack}
(1) 4-121   procedure Erase; virtual;
              {-Erase a window and pop from stack}
(1) 4-125   procedure Select; virtual;
              {-Make a window the current one by pulling to the top of stack}
(1) 4-120   procedure DrawHidden(var Under : StackWindow);
              {-Draw a window, inserting it in the stack underneath window Under}
(1) 4-121   procedure EraseHidden;
              {-Erase a window even if it's hidden on the window stack}
(1) 4-125   function OnStack : Boolean;
              {-Return True if window is displayed on the window stack}
(1) 4-118   procedure ActivateWrite;
              {-Activate fastwriting to a hidden window}
(1) 4-119   procedure DeactivateWrite;
              {-Deactivate fastwriting and display changes}
(1) 4-123   constructor Load(var S : IdStream);
              {-Load from stream}
(1) 4-127   procedure Store(var S : IdStream);
              {-Store to stream}
```

OpWindow CommandWindow

```
(1) 4-148   constructor Init(X1, Y1, X2, Y2 : Byte; var CP : CommandProcessor;
                             UnitCode : Byte);
              {-Create a command window with default options and colors}
(1) 4-148   constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                                   Options : LongInt; var CP : CommandProcessor;
                                   UnitCode : Byte);
              {-Create a command window with custom options}
(1) 4-152   procedure Process; virtual;
              {-Process commands}
(1) 4-154   procedure ProcessAutoPilot(var CmdList; NumCmds : Word);
              {-Process commands, starting with those in CmdList}
(1) 4-147   procedure GetNextCommand; virtual;
              {-Called by Process to get next command from user}
(1) 4-155   procedure RequestHelp(HelpIndex : Word); virtual;
              {-Called by Process to request context-sensitive help}
```

(1) 4-155 procedure SetCommandProcessor(var CP : CommandProcessor);
{-Set command processor to use}
(1) 4-143 function GetCommandProcessor : CommandProcessorPtr;
{-Return pointer to window's command processor}
(1) 4-145 function GetLastCommand : Word;
{-Return last command entered by user}
(1) 4-158 procedure SetLastCommand(Cmd : Word);
{-Change last command}
(1) 4-146 function GetLastKey : Word;
{-Return last keystroke entered by user}
(1) 4-158 procedure SetHelpIndex(Index : Word);
{-Set help topic number for this window}
(1) 4-144 function GetHelpIndex : Word;
{-Return the help topic number for this window}
(1) 4-156 procedure SetErrorProc(EP : ErrorProc);
{-Set error handler}
(1) 4-147 procedure GotError(ErrorCode : Word; ErrorMsg : string);
{-To be called when an error occurs}
(1) 4-145 function GetLastError : Word; virtual;
{-Return last I/O error code}
(1) 4-151 function PeekLastError : Word;
{-Return last I/O error code without resetting internal variable}
(1) 4-143 function ClassifyError(ErrorCode : Word) : Byte;
{-Return the class of the specified error code}
(1) 4-157 procedure SetExitCommands(ExitSetPtr : ByteSetPtr);
{-Specify set of special exit commands}
(1) 4-144 function GetExitCommandPtr : ByteSetPtr;
{-Return pointer to set of special exit commands}
(1) 4-149 constructor Load(var S : IdStream);
{-Load from stream}
(1) 4-159 procedure Store(var S : IdStream);
{-Store to stream}

OpWindow VirtScreen

(1) 4-176 constructor Alloc(Height, Width : Word);
{-Create a virtual screen of the specified dimensions}
(1) 4-180 constructor Init(Height, Width : Word; BufSeg : Word);
{-Initialize a virtual screen with a preallocated buffer}
(1) 4-179 destructor Done; virtual;
{-Deallocate a virtual screen}
(1) 4-175 procedure Activate;
{-Activate fast write and read routines on virtual screen}
(1) 4-179 procedure Deactivate;
{-Deactivate fast access to virtual screen}
(1) 4-176 procedure Clear(Attr : Byte; Fill : Char);
{-Clear virtual screen}
(1) 4-185 procedure WriteTo(S : string; Row, Col : Word; Attr : Byte);
{-Write a string to virtual screen}
(1) 4-182 procedure ReadFrom(Number : Byte; Row, Col : Word; var S : string);
{-Read number characters from the virtual screen starting at Row, Col}
(1) 4-182 procedure ScrollHoriz(Cols : Integer; Attr : Byte; Fill : Char);
{-Scroll contents horizontally}
(1) 4-183 procedure ScrollVert(Rows : Integer; Attr : Byte; Fill : Char);
{-Scroll contents vertically}
(1) 4-177 procedure CopyFromWindow(Row, Col : Word);
{-Copy current window to position <Row,Col> on virtual screen}
(1) 4-178 procedure CopyToWindow(Row, Col : Word);
{-Copy from position <Row,Col> on virtual screen to current window}
(1) 4-177 procedure CopyFromScreen(Row, Col : Word);
{-Copy physical screen to position <Row,Col> on virtual screen}
(1) 4-178 procedure CopyToScreen(Row, Col : Word);
{-Copy from position <Row,Col> on virtual screen to physical screen}

(1) 4-185 `function vRows : Word;`
`{-Return number of rows in virtual screen}`
(1) 4-184 `function vCols : Word;`
`{-Return number of columns in virtual screen}`
(1) 4-181 `constructor Load(var S : IdStream);`
`{-Load from stream}`
(1) 4-184 `procedure Store(var S : IdStream);`
`{-Store to stream}`

OpWindow WindowStack

(1) 4-133 `constructor Init(Max : Word);`
`{-Allocate stack and initialize}`
(1) 4-132 `destructor Done; virtual;`
`{-Clean up and dispose}`
(1) 4-137 `function UnstackTop : WindowPtr;`
`{-Erase and unstack window at top of stack}`
(1) 4-137 `function TopWindow : WindowPtr;`
`{-Return a pointer to the window at top of stack}`
(1) 4-133 `function EvaluateStackPos(X, Y : Byte) : WindowPtr;`
`{-Evaluate position and return pointer to indicated window, if any}`
(1) 4-132 `function EvaluateStackMousePos : WindowPtr;`
`{-Evaluate mouse position using EvaluatePos}`
(1) 4-134 `constructor Load(var S : IdStream);`
`{-Load from stream}`
(1) 4-135 `procedure Store(var S : IdStream);`
`{-Store to stream. Stores all currently stacked windows}`

OpWindow LoadableColorSet

(1) 4-162 `constructor Init(var Colors : ColorSet);`
`{-Initialize this color set from another one}`
(1) 4-162 `constructor Load(var S : IdStream);`
`{-Load a color set from a stream}`
(1) 4-163 `procedure Store(var S : IdStream);`
`{-Store a color set in a stream}`

OpWindow PackedWindow

(1) 4-167 `constructor Init(XL, YL, XH, YH : Byte);`
`{-Create a packed window}`
(1) 4-168 `constructor InitFromMemory(P : Pointer);`
`{-Initialize a packed window from one stored to disk with Write and bound into the EXE file with BINOBJ}`
(1) 4-167 `destructor Done; virtual;`
`{-Deallocate the packed screen}`
(1) 4-170 `constructor Read(FName : string);`
`{-Read the packed window stored in FName into memory}`
(1) 4-172 `function Write(FName : string) : Word;`
`{-Store the packed window in FName}`
(1) 4-169 `constructor Load(var S : IdStream);`
`{-Load a packed window from a stream}`
(1) 4-171 `procedure Store(var S : IdStream);`
`{-Store a packed window in a stream}`
(1) 4-166 `procedure Display;`
`{-Display the packed window}`
(1) 4-166 `procedure DisplayAt(Row, Col : Word);`
`{-Display the packed window at Row,Col. If necessary, the coordinates are adjusted to allow it to fit on the screen.}`
(1) 4-170 `procedure MapColors;`
`{-Map the colors of the packed window for improved appearance on mono/B&W displays}`

OpPick **PickList**

```
(1) 4-218   constructor Init(X1, Y1, X2, Y2 : Byte; ItemWidth : Byte;
                             NumItems : Word; StringProc : pkStringProc;
                             Orientation : pkGenlProc;
                             CommandHandler : pkGenlProc);
              {-Initialize a pick window}
(1) 4-220   constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                                   Options : LongInt; ItemWidth : Byte;
                                   NumItems : Word; StringProc : pkStringProc;
                                   Orientation : pkGenlProc;
                                   CommandHandler : pkGenlProc);
              {-Initialize a pick window with custom window options}
(1) 4-219   constructor InitAbstract(X1, Y1, X2, Y2 : Byte;
                                     var Colors : ColorSet; Options : LongInt;
                                     ItemWidth : Byte; NumItems : Word;
                                     Orientation : pkGenlProc;
                                     CommandHandler : pkGenlProc);
               {-Constructor to be called by derived types that override
                 the ItemString method}
(1) 4-221   constructor InitDeluxe(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                                   Options : LongInt; ItemWidth : Byte;
                                   NumItems : Word; StringProc : pkStringProc;
                                   Orientation : pkGenlProc;
                                   CommandHandler : pkGenlProc;
                                   PickOptions : Word);
              {-Initialize a pick window with custom window and pick options}
(1) 4-219   constructor InitAbstractDeluxe(X1, Y1, X2, Y2 : Byte;
                                           var Colors : ColorSet;
                                           Options : LongInt; ItemWidth : Byte;
                                           NumItems : Word;
                                           Orientation : pkGenlProc;
                                           CommandHandler : pkGenlProc;
                                           PickOptions : Word);
               {-Constructor to be called by derived types that override the
                 ItemString method, with custom pick options}
(1) 4-214   destructor Done; virtual;
              {-Dispose of PickList}
(1) 4-208   procedure AddMoreHeader(PromptStr : String80;
                                    Posn : HeaderPosType;
                                    UpStr, DnStr, SepStr : String10;
                                    UpPos, DnPos, SepPos : Byte);
              {-Add a specialized more header to indicate more pick items}
(1) 4-209   procedure AddSearchHeader(MaxLen : Byte; Posn : HeaderPosType);
              {-Add a specialized search string header}
(1) 4-240   procedure UpdateContents; virtual;
              {-Redraw the complete pick window}
(1) 4-230   procedure Process; virtual;
              {-Process pick commands leading to a choice}
(1) 4-224   procedure ItemString(Item : Word; Mode : pkMode;
                                 var IType : pkItemType;
                                 var IString : String); virtual;
              {-Supplies each item string when the list is displayed or searched}
(1) 4-213   procedure CompleteItemString(Item : Word; Mode : pkMode;
                                         var IType : pkItemType;
                                         var IString : String); virtual;
              {-Supplies, marks, and pads item string}
(1) 4-230   procedure PreMove; virtual;
              {-Called just prior to getting each keyboard command}
(1) 4-224   function ItemSearch : Boolean; virtual;
              {-Called to search list each time an alphanumeric character is
                entered}
(1) 4-232   procedure SetInitialChoice(Choice : Word);
              {-Set initial choice without scrolling if possible}
(1) 4-231   procedure SetChoice(Choice, FirstChoice : Word);
              {-Set initial choice with scrolling control}
(1) 4-232   procedure SetDefaultChoice(Choice : Word);
              {-Set the default choice number returned by GetDefaultChoice}
```

OPPICK

(1) 4-214
```
function GetDefaultChoice : Word;
  {-Return default choice number}
```
(1) 4-216
```
function GetLastChoice : Word;
  {-Return last selected item number}
```
(1) 4-215
```
function GetItemString(Item : Word) : string;
  {-Return specified item string in search format}
```
(1) 4-216
```
function GetLastChoiceString : string;
  {-Return last choice in search format}
```
(1) 4-240
```
procedure WhereSelect(var X, Y : Byte);
  {-Return window-relative coordinates of selected item}
```
(1) 4-240
```
procedure WhereSelectAbs(var X, Y : Byte);
  {-Return absolute coordinates of selected item}
```
(1) 4-217
```
function GetOrientation : Byte;
  {-Return the orientation code}
```
(1) 4-215
```
function GetItemCols : Byte;
  {-Return number of visible item columns}
```
(1) 4-216
```
function GetNumItems : Word;
  {-Return number of pick items}
```
(1) 4-229
```
procedure pkOptionsOn(OptionFlags : Word);
  {-Activate multiple options}
```
(1) 4-229
```
procedure pkOptionsOff(OptionFlags : Word);
  {-Deactivate multiple options}
```
(1) 4-229
```
function pkOptionsAreOn(OptionFlags : Word) : Boolean;
  {-Return True if all specified options are on}
```
(1) 4-236
```
procedure SetRowLimits(MinRows, MaxRows : Word);
  {-Set values for min and max rows}
```
(1) 4-233
```
procedure SetPadSize(LeftPad, RightPad : Byte);
  {-Set number of characters for padding of pick items}
```
(1) 4-233
```
procedure SetMoveProc(MP : pkGenlProc);
  {-Set user move procedure}
```
(1) 4-236
```
procedure SetSearchMode(SF : pkSearchFunc);
  {-Set a different search mode}
```
(1) 4-236
```
procedure SetSearchStart(SStart : Byte);
  {-Set start offset for string searching}
```
(1) 4-211
```
procedure ChangeNumItems(NumItems : Word);
  {-Change the number of items to display}
```
(1) 4-210
```
procedure ChangeItemWidth(ItemWidth : Byte);
  {-Change the item width}
```
(1) 4-212
```
procedure ChangeStringProc(StringProc : pkStringProc);
  {-Change the string function}
```
(1) 4-211
```
procedure ChangeOrientation(Orientation : pkGenlProc);
  {-Change the orientation}
```
(1) 4-210
```
procedure ChangeCommandHandler(CommandHandler : pkGenlProc);
  {-Change between single and multiple choice}
```
(1) 4-228
```
procedure OptimizeSize;
  {-Resize window to surround items}
```
(1) 4-234
```
procedure SetPickAttr(ItemType : pkItemType; Selected : Boolean;
                      Color, Mono : Byte);
  {-Set one group of colors -- disables flexwriting}
```
(1) 4-235
```
procedure SetPickFlex(ItemType : pkItemType; Selected : Boolean;
                      var Color, Mono : FlexAttrs);
  {-Set one group of colors for flexwriting}
```
(1) 4-237
```
procedure SetSelectMarker(Left, Right : String);
  {-Define strings to mark selected items}
```
(1) 4-209
```
procedure AllocateSelectSet(MaxItems : Word);
  {-Allocate the select set for the maximum items}
```
(1) 4-212
```
procedure ClearSelected;
  {-Clear selected items}
```
(1) 4-226
```
procedure MarkAllSelected;
  {-Mark all items selected}
```
(1) 4-231
```
procedure SelectItem(Item : Word); virtual;
  {-Select an item}
```
(1) 4-213
```
procedure DeselectItem(Item : Word); virtual;
  {-Deselect an item}
```

(1) 4-224 `function ItemIsSelected(Item : Word) : Boolean;`
`{-Return True if item is selected}`

(1) 4-217 `function GetSelectedCount : Word;`
`{-Return number of items selected}`

(1) 4-223 `procedure InitSequence(var Item : Word);`
`{-Prepare to get first selected item}`

(1) 4-218 `function HaveSelected(Item : Word) : Boolean;`
`{-Return True if there is a selected item}`

(1) 4-226 `procedure NextSelected(var Item : Word);`
`{-Position Item at the next selected item}`

(1) 4-223 `function IsProtected(Item : Word) : Boolean;`
`{-Return True if item is protected from selection}`

(1) 4-227 `function NumSelectableItems : Word;`
`{-Return number of items that are not protected}`

(1) 4-225 `constructor Load(var S : IdStream);`
`{-Load a pick list from a stream}`

(1) 4-238 `procedure Store(var S : IdStream);`
`{-Store a pick list in a stream}`

OpPick Procedures and Functions

(1) 4-199 `function PickNoSearch(P : PickListPtr) : Boolean;`
`{-No searching mode}`

(1) 4-199 `function PickStringSearch(P : PickListPtr) : Boolean;`
`{-String searching mode}`

(1) 4-199 `function PickAltStringSearch(P : PickListPtr) : Boolean;`
`{-Alternate string searching mode}`

(1) 4-199 `function PickCharSearch(P : PickListPtr) : Boolean;`
`{-Character searching mode}`

(1) 4-199 `function PickCharExit(P : PickListPtr) : Boolean;`
`{-Character searching with exit on match}`

(1) 4-199 `function PickAnyCharExit(P : PickListPtr) : Boolean;`
`{-Exit on any alpha character. No highlight repositioning}`

(1) 4-194 `procedure PickVertical(P : PickListPtr);`
`{-Initialize pick for items arranged vertically}`

(1) 4-194 `procedure PickHorizontal(P : PickListPtr);`
`{-Initialize pick for items arranged horizontally}`

(1) 4-194 `procedure PickSnaking(P : PickListPtr);`
`{-Initialize pick for items arranged vertically, snaking}`

(1) 4-188 `procedure SingleChoice(P : PickListPtr);`
`{-Initialize for a single choice picklist}`

(1) 4-188 `procedure MultipleChoice(P : PickListPtr);`
`{-Initialize for a multiple choice picklist}`

(1) 4-233 `procedure NoMoveAction(P : PickListPtr);`
`{-User hook routine called just before getting each pick command}`

OpDir DirList

(1) 4-265
```
constructor Init(X1, Y1, X2, Y2 : Byte; HeapToUse : Word;
                 PickOrientation : pkGenlProc;
                 CommandInit : diInitCommandProc);
```
`{-Initialize a directory list}`

(1) 4-266
```
constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                       Options : LongInt; HeapToUse : Word;
                       PickOrientation : pkGenlProc;
                       CommandInit : diInitCommandProc);
```
`{-Initialize a directory list with custom window options}`

(1) 4-256 `destructor Done; virtual;`
`{-Dispose of directory list}`

(1) 4-251
```
procedure AddMaskHeader(FullPath : Boolean; PadSize : Byte;
                        MaxLen : Byte; Posn : HeaderPosType);
```
`{-Add a header to display the search mask}`

(1) 4-274 `procedure SetMask(Mask : String; FileAttr : Byte);`
`{-Set search mask and attribute and reset list}`

(1) 4-271 procedure ResetList;
{-Force reread of directory when it may have changed}
(1) 4-268 procedure PreloadDirList;
{-Read directory in advance, in order to optimize size before drawing}
(1) 4-268 procedure Process; virtual;
{-Process pick commands leading to a file choice}
(1) 4-281 procedure UpdateContents; virtual;
{-Redraw the contents of the directory window}
(1) 4-278 procedure SortList; virtual;
{-Sort the list of directory entries}
(1) 4-269 function Reject(var X : DirRec) : Boolean; virtual;
{-Return True to exclude a particular entry from pick}
(1) 4-257 procedure Format(var X : DirRec; var pkCat : Byte;
var S : String); virtual;
{-Return a formatted string and category for a directory entry}
(1) 4-258 function GetFileName(Mask : String; FileAttr : Byte;
var Name : PathStr) : Word;
{-Set a new mask, pick a name and return. Returns status}
(1) 4-253 procedure ChangeDirectory(Mask : String; ChangeCode : Byte);
{-Change to specified directory, or let user browse around to do so}
(1) 4-260 function GetMatchingFileCount : Word;
{-Return number of files matched in last directory}
(1) 4-260 function GetMatchingSize(ClusterSize : Word) : LongInt;
{-Return number of bytes in all matched files}
(1) 4-264 function GetSelectedFile : FileStr;
{-Return 12-character name of selected file}
(1) 4-264 function GetSelectedPath : PathStr;
{-Return full pathname of selected file}
(1) 4-263 function GetSelectedDir : DirStr;
{-Return directory of selected file}
(1) 4-263 procedure GetSelectedDirRec(var D : DirRec);
{-Return full DirRec for selected file}
(1) 4-272 function SelectedFileInCurDir : Boolean;
{-Return true if selected file is in current directory}
(1) 4-270 procedure RemoveSelectedFile;
{-Remove current file from pick list}
(1) 4-255 procedure diOptionsOn(OptionFlags : Word);
{-Activate multiple options}
(1) 4-255 procedure diOptionsOff(OptionFlags : Word);
{-Deactivate multiple options}
(1) 4-254 function diOptionsAreOn(OptionFlags : Word) : Boolean;
{-Return True if all specified options are on}
(1) 4-256 function FileCapacity : Word;
{-Return the maximum number of files list can hold}
(1) 4-278 procedure SetNameFormat;
{-Initialize for compact nnnnnnnn.xxx format}
(1) 4-278 procedure SetNameSizeFormat(DirStr : String);
{-Initialize for nnnnnnn.xxx ssssssss format}
(1) 4-278 procedure SetNameSizeKFormat(DirStr : String);
{-Initialize for nnnnnnn.xxx ssssssk format}
(1) 4-278 procedure SetNameTimeFormat(DatePicture, TimePicture : String);
{-Initialize for name date time format}
(1) 4-278 procedure SetNameSizeTimeFormat(DirStr,
DatePicture, TimePicture : String);
{-Initialize for name size date time format}
(1) 4-278 procedure SetNameSizeKTimeFormat(DirStr,
DatePicture, TimePicture : String);
{-Initialize for name size date time format}
(1) 4-278 procedure SetAllFormat(DirStr, DatePicture, TimePicture : String);
{-Initialize for name size date time attr format}
(1) 4-278 procedure SetUserFormat(MaxFmatLen : Byte; FF : diFormatProc);
{-Set up for a user-defined format function}

(1) 4-273
```
procedure SetFileAttr(Color, Mono : Byte);
  {-Set video attributes for files}
```
(1) 4-273
```
procedure SetDirAttr(Color, Mono : Byte);
  {-Set video attributes for directories}
```
(1) 4-276
```
procedure SetSelectFileAttr(Color, Mono : Byte);
  {-Set video attributes for selected file}
```
(1) 4-276
```
procedure SetSelectDirAttr(Color, Mono : Byte);
  {-Set video attributes for selected directory}
```
(1) 4-277
```
procedure SetSortOrder(SP : diSortProc);
  {-Set alternate routine for sorting}
```
(1) 4-275
```
procedure SetRejectFunc(RF : diRejectFunc);
  {-Set alternate reject function for file selection}
```
(1) 4-252
```
procedure AddRejectString(S : String);
  {-Add a string to the set to reject}
```
(1) 4-254
```
procedure ClearRejectStrings;
  {-Clear the set of strings to reject}
```
(1) 4-262
```
function GetMultiFile(N : Word) : FileStr;
  {-Return 12-character name of specified file}
```
(1) 4-262
```
function GetMultiPath(N : Word) : PathStr;
  {-Return full pathname of specified file}
```
(1) 4-261
```
procedure GetMultiDirRec(N : Word; var D : DirRec);
  {-Return full DirRec for specified file}
```
(1) 4-265
```
function GetSelectedSize(ClusterSize : Word) : LongInt;
  {-Return total size of all selected files}
```
(1) 4-269
```
procedure RemoveMultiFile(N : Word);
  {-Remove specified file from list}
```
(1) 4-270
```
procedure RemoveSelectedFiles;
  {-Remove all selected files from list}
```
(1) 4-261
```
function GetMultiDirPtr(N : Word) : DirRecPtr;
  {-Return pointer to DirRec for specified file}
```
(1) 4-257
```
procedure FormatMultiFile(N : Word);
  {-Rebuild the format string for specified file}
```
(1) 4-272
```
procedure SelectItem(N : Word); virtual;
  {-Select an item. Overrides PickList.SelectItem to avoid
    selecting directories}
```
(1) 4-267
```
constructor Load(var S : IdStream);
  {-Load a list from a stream}
```
(1) 4-279
```
procedure Store(var S : IdStream);
  {-Store a list in a stream}
```

OpDir PathList

(1) 4-287
```
constructor Init(X1, Y1, X2, Y2 : Byte; HeapToUse : Word;
                 PickOrientation : pkGenlProc;
                 CommandInit : paInitCommandProc);
  {-Initialize a path list}
```
(1) 4-288
```
constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                       Options : LongInt; HeapToUse : Word;
                       PickOrientation : pkGenlProc;
                       CommandInit : paInitCommandProc);
  {-Initialize a path list with custom window options}
```
(1) 4-285
```
destructor Done; virtual;
  {-Dispose of path list}
```
(1) 4-293
```
procedure SetDrive(Drive : Char);
  {-Specify drive for path list}
```
(1) 4-292
```
procedure ResetList;
  {-Force reread of directories when they may have changed}
```
(1) 4-296
```
procedure UpdateContents; virtual;
  {-Redraw the complete pick window}
```
(1) 4-291
```
procedure Process; virtual;
  {-Display list if any, get one selection}
```
(1) 4-291
```
procedure PreloadPathList;
  {-Read paths in advance, in order to optimize size before drawing}
```

(1) 4-286 function GetPathName(Drive : Char) : PathStr;
{-Set drive, pick a path, and return it. Returns empty string if error.}
(1) 4-287 function GetSelectedPath : PathStr;
{-Return selected pathname}
(1) 4-290 procedure paOptionsOn(OptionFlags : Word);
{-Activate multiple options}
(1) 4-290 procedure paOptionsOff(OptionFlags : Word);
{-Deactivate multiple options}
(1) 4-290 function paOptionsAreOn(OptionFlags : Word) : Boolean;
{-Return True if all specified options are on}
(1) 4-292 procedure SetClusterSize(Size : Word);
{-Set cluster size for adjusting files to DOS boundaries}
(1) 4-293 procedure SetDrawingChars(P : PathDrawArray);
{-Set the characters used for line graphics}
(1) 4-294 procedure SetLineDrawMode;
{-Display hierarchy with line draw characters (default)}
(1) 4-294 procedure SetFullPathMode;
{-Display hierarchy with full path names}
(1) 4-295 procedure SetPathAttr(Color, Mono : Byte);
{-Set video attributes for paths}
(1) 4-295 procedure SetSelectPathAttr(Color, Mono : Byte);
{-Set video attributes for selected path}
(1) 4-285 function GetMultiPath(N : Word) : PathStr;
{-Return pathname of specified path}
(1) 4-289 constructor Load(var S : IdStream);
{-Load a list from a stream}
(1) 4-296 procedure Store(var S : IdStream);
{-Store a list in a stream}

OpDir Procedures and Functions

(1) 4-242 procedure SingleFile(DirPtr : DirListPtr;
var StringProc : pkStringProc;
var CommandInitProc : pkGenlProc);
{-Initialize for picking single files}
(1) 4-242 procedure MultipleFile(DirPtr : DirListPtr;
var StringProc : pkStringProc;
var CommandInitProc : pkGenlProc);
{-Initialize for picking multiple files}
(1) 4-246 procedure SortNone(DirPtr : DirListPtr);
{-No sort}
(1) 4-246 procedure SortName(DirPtr : DirListPtr);
{-Sort alphabetically by name, then by extension}
(1) 4-246 procedure SortDirName(DirPtr : DirListPtr);
{-Sort directories first, then alphabetically by name and extension}
(1) 4-246 procedure SortSize(DirPtr : DirListPtr);
{-Sort by size, then by name and extension}
(1) 4-246 procedure SortTime(DirPtr : DirListPtr);
{-Sort by date, then by name/extension}
(1) 4-247 procedure NameFormat(var D : DirRec; var pkCat : Byte;
var S : String; DirPtr : DirListPtr);
{-Simple default format for directory lists}
(1) 4-247 procedure NameSizeKFormat(var D : DirRec; var pkCat : Byte;
var S : String; DirPtr : DirListPtr);
{-Show name and size in Kilobytes}
(1) 4-247 procedure NameSizeFormat(var D : DirRec; var pkCat : Byte;
var S : String; DirPtr : DirListPtr);
{-Show name and size in bytes}
(1) 4-247 procedure NameTimeFormat(var D : DirRec; var pkCat : Byte;
var S : String; DirPtr : DirListPtr);
{-Show name and date/time}

```
(1) 4-247    procedure NameSizeKTimeFormat(var D : DirRec; var pkCat : Byte;
                                           var S : String; DirPtr : DirListPtr);
               {-Show name, size(K), time}
(1) 4-247    procedure NameSizeTimeFormat(var D : DirRec; var pkCat : Byte;
                                          var S : String; DirPtr : DirListPtr);
               {-Show name, size, time}
(1) 4-247    procedure AllFormat(var D : DirRec; var pkCat : Byte;
                                 var S : String; DirPtr : DirListPtr);
               {-Show name, size, time, attr}
(1) 4-245    function RejectNone(var D : DirRec; DirPtr : DirListPtr) : Boolean;
               {-Don't reject any files}
(1) 4-245    function RejectExtensions(var D : DirRec;
                                       DirPtr : DirListPtr) : Boolean;
               {-Reject files with extensions in RejectSet}
(1) 4-245    function RejectFiles(var D : DirRec; DirPtr : DirListPtr) : Boolean;
               {-Reject all but subdirectories}
(1) 4-283    procedure SinglePath(PathPtr : PathListPtr;
                                  var StringProc : pkStringProc;
                                  var CommandInitProc : pkGenlProc);
               {-Initialize for picking single paths}
(1) 4-283    procedure MultiplePath(PathPtr : PathListPtr;
                                    var StringProc : pkStringProc;
                                    var CommandInitProc : pkGenlProc);
               {-Initialize for picking multiple paths}
```

OpMenu MenuItemNode

```
(1) 4-318    procedure Protect;
               {-Protect item}
(1) 4-319    procedure Unprotect;
               {-Unprotect item}
(1) 4-315    function ChangeName(Name : String) : Boolean;
               {-Change the name of item}
(1) 4-315    procedure ChangeKey(Key : LongInt);
               {-Change the item key}
(1) 4-314    function ChangeHelp(Help : String) : Boolean;
               {-Change the help string}
(1) 4-315    procedure ChangeSelectPos(SelectPos : Byte);
               {-Change the select character offset}
(1) 4-314    procedure ChangeDisplayPos(DisplayPos : Byte);
               {-Change the display offset}
(1) 4-316    procedure ChangeSubPtr(SPtr : SubMenuPtr);
               {-Change submenu pointer of item}
(1) 4-316    procedure DisplayInfo(var DisplayPos, SelectPos : Byte;
                                   var S : String);
               {-Return string and display offsets for item}
(1) 4-317    function HelpString : String;
               {-Return help string for item}
(1) 4-318    function ItemKey : LongInt;
               {-Return menu key for item}
(1) 4-319    function SubPtr : SubMenuPtr;
               {-Return submenu pointer for item}
(1) 4-317    function IsProtected : Boolean;
               {-Return True if item is protected}
(1) 4-318    function SelectChar : Char;
               {-Return alpha selection char (uppercase)}
(1) 4-317    function GetDisplayPos : Byte;
               {-Return just the display offset}
```

OpMenu Menu

```
(1) 4-338    constructor Init(X1, Y1, X2, Y2 : Byte; Orient : mnOrientation);
               {-Create a menu with default options and colors}
```

OPMENU

(1) 4-339
```
constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                       Options : LongInt; Orient : mnOrientation);
  {-Create a menu with custom window options}
```
(1) 4-344
```
procedure mnOptionsOn(OptionFlags : Word);
  {-Activate multiple options}
```
(1) 4-344
```
procedure mnOptionsOff(OptionFlags : Word);
  {-Deactivate multiple options}
```
(1) 4-344
```
function mnOptionsAreOn(OptionFlags : Word) : Boolean;
  {-Return True if all specified options are on}
```
(1) 4-325
```
procedure AddItem(Name : String; DisplayPos, SelectPos : Byte;
                  Key : LongInt);
  {-Add item to current item list, allocating string space on heap}
```
(1) 4-325
```
procedure AddItemPtr(var Name : String; DisplayPos, SelectPos : Byte;
                     Key : LongInt);
  {-Add item to current item list, not allocating string space on
    heap}
```
(1) 4-325
```
procedure AddItemHelp(Name : String; DisplayPos, SelectPos : Byte;
                      Key : LongInt; Help : String);
  {-Add item to current item list, allocating string space on heap}
```
(1) 4-325
```
procedure AddItemHelpPtr(var Name : String;
                         DisplayPos, SelectPos : Byte;
                         Key : LongInt; var Help : String);
  {-Add item to current item list, not allocating string space on
    heap}
```
(1) 4-328
```
procedure AddSeparator(FirstChar, SpanChar, LastChar : Char;
                       DisplayPos : Byte);
  {-Add a separator bar at offset DisplayPos within the window}
```
(1) 4-328
```
procedure AddSeparatorColor(FirstChar, SpanChar, LastChar : Char;
                            DisplayPos : Byte;
                            AttrColor, AttrMono : Byte);
  {-Add a separator bar at offset DisplayPos within the window with
    specified colors}
```
(1) 4-329
```
procedure AddSubMenu(X1, Y1, X2, Y2 : Byte; Orient : mnOrientation);
  {-Add an unframed submenu with options and colors cloned from main
    menu}
```
(1) 4-324
```
procedure AddFramedSubMenu(X1, Y1, X2, Y2 : Byte;
                           Orient : mnOrientation; FA : FrameArray);
  {-Add a framed submenu with options and colors cloned from main
    menu}
```
(1) 4-330
```
procedure AddSubMenuCustom(X1, Y1, X2, Y2 : Byte;
                           var Colors : ColorSet; Options : LongInt;
                           Orient : mnOrientation; FA : FrameArray);
  {-Add a submenu with custom window options}
```
(1) 4-341
```
procedure ItemsDone;
  {-Terminate the current item list. Set first item as default.}
```
(1) 4-342
```
procedure ItemsDoneCustom(OptimizeSize : Boolean;
                          DefaultKey : LongInt);
  {-Terminate the current item list and customize size and default
    key}
```
(1) 4-353
```
procedure SetNormAttr(Color, Mono : Byte);
  {-Set attributes for normal item text}
```
(1) 4-353
```
procedure SetSelectAttr(Color, Mono : Byte);
  {-Set attributes for selected item text}
```
(1) 4-353
```
procedure SetHighAttr(Color, Mono : Byte);
  {-Set attributes for pick highlight text}
```
(1) 4-353
```
procedure SetProtectAttr(Color, Mono : Byte);
  {-Set attributes for protected item text}
```
(1) 4-353
```
procedure SetFrameAttr(Color, Mono : Byte);
  {-Set frame attributes of all windows in menu system}
```
(1) 4-353
```
procedure SetHeaderAttr(Color, Mono : Byte);
  {-Set header attributes of all windows in menu system}
```
(1) 4-353
```
procedure SetShadowAttr(Color, Mono : Byte);
  {-Set shadow attributes of all windows in menu system}
```
(1) 4-354
```
procedure SetLeftPad(LeftPad : Byte);
  {-Set the number of characters to left pad each item when
    displayed}
```

```
(1) 4-354    procedure SetSelectMarker(Left, Right : String);
               {-Define strings to mark selected items}
(1) 4-320    procedure AddHeader(S : string; Posn : HeaderPosType);
               {-Add a standard header}
(1) 4-320    procedure AddHeaderColor(S : string; Posn : HeaderPosType;
                                      AttrColor, AttrMono : Byte);
               {-Add a standard header with custom color}
(1) 4-320    procedure AddCustomHeader(S : string; Posn : FrameCornerType;
                                       DX, DY : Integer;
                                       AttrColor, AttrMono : Byte);
               {-Add a custom header}
(1) 4-329    procedure AddShadow(Posn : ShadowPosType; DrawType : ShadowDrawType);
               {-Add a standard shadow}
(1) 4-320    procedure AddShadowColor(Posn : ShadowPosType;
                                      DrawType : ShadowDrawType;
                                      AttrColor, AttrMono : Byte);
               {-Add a standard shadow with custom color}
(1) 4-320    procedure EnableExplosions(StageDelay : Word);
               {-Enable exploding windows with StageDelay milliseconds per stage}
(1) 4-320    procedure EnableNormalOpen;
               {-Disable exploding windows or alternate draw routines}
(1) 4-350    procedure SetAllNormAttr(Color, Mono : Byte);
               {-Set normal attributes of all items in menu system}
(1) 4-350    procedure SetAllSelectAttr(Color, Mono : Byte);
               {-Set select attributes of all items in menu system}
(1) 4-350    procedure SetAllHighAttr(Color, Mono : Byte);
               {-Set highlight attributes of all items in menu system}
(1) 4-350    procedure SetAllProtectAttr(Color, Mono : Byte);
               {-Set protect attributes of all items in menu system}
(1) 4-350    procedure SetAllFrameAttr(Color, Mono : Byte);
               {-Set frame attributes of all windows in menu system}
(1) 4-350    procedure SetAllHeaderAttr(Color, Mono : Byte);
               {-Set header attributes of all windows in menu system}
(1) 4-350    procedure SetAllShadowAttr(Color, Mono : Byte);
               {-Set shadow attributes of all windows in menu system}
(1) 4-329    procedure AddShadows(MainMenu : Boolean; Posn : ShadowPosType;
                                  DrawType : ShadowDrawType);
               {-Add shadows to all submenus}
(1) 4-333    procedure EnableAllExplosions(StageDelay : Word);
               {-Enable exploding windows with StageDelay milliseconds per stage}
(1) 4-332    procedure EnableAllAlternateOpen(OpenProc, CloseProc : RawWindowProc;
                                              StageDelay : Word);
               {-Enable alternate draw and erase routines}
(1) 4-333    procedure EnableAllNormalOpen;
               {-Disable exploding windows or alternate draw routines}
(1) 4-351    procedure SetAllSelectMarker(Left, Right : String);
               {-Define strings to mark selected items in all submenus}
(1) 4-352    procedure SetHelpAttr(Color, Mono : Byte);
               {-Set the colors for the help line}
(1) 4-337    procedure GetHelpAttr(var Color, Mono : Byte);
               {-Get the colors stored for the help line}
(1) 4-358    procedure VisitAllItems(IProc : mnItemProc; var D);
               {-Call the specified procedure for all items in the menu system}
(1) 4-358    procedure VisitAllWindows(WProc : mnWindowProc; var D);
               {-Call the specified procedure for all submenus in the menu system}
(1) 4-351    procedure SetCurrentItemProc(IP : mnCurrentItemProc);
               {-Set procedure to be called whenever the active item is drawn}
(1) 4-352    procedure SetCustomStringProc(CP : mnCustomProc);
               {-Set item string customization procedure}
(1) 4-330    procedure CurrentItem(IPtr : MenuItemNodePtr); virtual;
               {-Called just after the currently selected item is drawn}
(1) 4-346    procedure Process; virtual;
               {-Process menu commands leading to a choice}
(1) 4-344    function MenuChoice : LongInt;
               {-Return the last menu choice}
```

```
(1) 4-336   procedure EraseCurrentSubMenu;
              {-Erase current submenu}
(1) 4-335   procedure EraseAllSubMenus(ShowSelect, ShowHighlight : Boolean);
              {-Erase all submenus, leave main menu showing in specified state}
(1) 4-336   procedure EraseClear;
              {-Erase menu and clear active submenus for next draw}
(1) 4-348   procedure SelectItem(Key : LongInt);
              {-Select specified item and update screen}
(1) 4-350   procedure SelectSubMenu(Key : LongInt);
              {-Select current item of submenu containing Key and update screen}
(1) 4-349   function SelectItemByPos(X, Y : Byte) : Boolean;
              {-Select visible item (if any) at absolute position and update
                screen}
(1) 4-347   procedure Redraw;
              {-Erase and redraw menu system, presumably after changing an item}
(1) 4-346   procedure ProtectItem(Key : LongInt);
              {-Protect specified item. No screen update}
(1) 4-357   procedure UnprotectItem(Key : LongInt);
              {-Unprotect specified item. No screen update}
(1) 4-331   procedure DefaultItem(Key : LongInt);
              {-Make item Key the default of its submenu. No screen update}
(1) 4-331   procedure DefaultPath(Key : LongInt);
              {-Select path to specified item. No screen update}
(1) 4-323   function ActiveListPtr : MenuItemListPtr;
              {-Return pointer to active item list}
(1) 4-323   function ActiveWinPtr : RawWindowPtr;
              {-Return pointer to active raw window}
(1) 4-323   function ActiveSubPtr : SubMenuPtr;
              {-Return pointer to active submenu}
(1) 4-323   function ActiveItemPtr : MenuItemNodePtr;
              {-Return pointer to currently active item}
(1) 4-337   function FindItem(Key : LongInt) : MenuItemNodePtr;
              {-Return pointer to item node}
(1) 4-343   constructor Load(var S : IdStream);
              {-Load a list from a stream}
(1) 4-355   procedure Store(var S : IdStream);
              {-Store a list in a stream}
(1) 4-332   destructor Done; virtual;
              {-Destroy a menu}
(1) 4-334   procedure Erase; virtual;
              {-Erase menu system. Remembers previous submenus for next draw}
(1) 4-341   function IsCurrent : Boolean; virtual;
              {-Return True if menu system is the current window}
(1) 4-345   procedure MoveWindow(DX, DY : Integer); virtual;
              {-Move a menu}
(1) 4-357   procedure UpdateScreenSize;
              {-Update window size, position, and limits for current screen size}
```

OpMenu Procedures and Functions

```
(1) 4-352   procedure NoCustomization(var Name : String; Key : LongInt;
                                      Selected, Highlighted : Boolean;
                                      WPtr : RawWindowPtr);
              {-Default string customization routine that does nothing}
(1) 4-351   procedure NoCurrItemAction(CurrentItem : MenuItemNodePtr;
                                       MPtr : MenuPtr);
              {-Default active item move routine that does nothing}
```

OpHelp AbstractHelpWindow

```
(2) 4-390   constructor Init(X1, Y1, X2, Y2 : Byte;
                             HelpFile : PathStr;
                             PickOrientation : pkGenlProc);
              {-Initialize help window and open help file}
```

```
(2) 4-394     constructor InitMem(X1, Y1, X2, Y2 : Byte;
                                  HelpAddr : Pointer;
                                  PickOrientation : pkGenlProc);
                {-Initialize help window using help bound into code}
(2) 4-391     constructor InitCustom(X1, Y1, X2, Y2 : Byte;
                                     var Colors : ColorSet;
                                     Options : LongInt;
                                     HelpFile : PathStr;
                                     PickOrientation : pkGenlProc);
                {-Initialize help window and open help file}
(2) 4-392     constructor InitDeluxe(X1, Y1, X2, Y2 : Byte;
                                     var Colors : ColorSet;
                                     Options : LongInt;
                                     HelpFile : PathStr;
                                     PickOrientation : pkGenlProc;
                                     HelpOptions : Word);
                {-Initialize help window and open help file}
(2) 4-394     constructor InitMemCustom(X1, Y1, X2, Y2 : Byte;
                                        var Colors : ColorSet;
                                        Options : LongInt;
                                        HelpAddr : Pointer;
                                        PickOrientation : pkGenlProc);
                {-Initialize help window using help bound into code}
(2) 4-395     constructor InitMemDeluxe(X1, Y1, X2, Y2 : Byte;
                                        var Colors : ColorSet;
                                        Options : LongInt;
                                        HelpAddr : Pointer;
                                        PickOrientation : pkGenlProc;
                                        HelpOptions : Word);
                {-Initialize help window using help bound into code}
(2) 4-385     destructor Done; virtual;
                {-Close up and dispose of help window}
(2) 4-396     procedure ItemString(Item : Word; Mode : pkMode;
                                   var IType : pkItemType;
                                   var IString : String); virtual;
                {-Supplies each item string when the list is displayed or searched}
(2) 4-386     procedure Draw; virtual;
                {-Choose appropriate frame and draw help window or pick index}
(2) 4-405     procedure UpdateContents; virtual;
                {-Redraw the help window in pick or help mode}
(2) 4-398     procedure Process; virtual;
                {-Draw help index or current topic}
(2) 4-402     procedure SetTopic(Topic : Word);
                {-Specify which topic is displayed next. 0 for index}
(2) 4-388     function GetTopicChoice : Word;
                {-Return topic chosen by pick}
(2) 4-390     function InHelpMode : Boolean;
                {-Return True if window shows help, not pick list}
(2) 4-405     function TopicStackPtr : StaticQueuePtr;
                {-Return a pointer to the topic stack}
(2) 4-385     function CurrentTopic : Word;
                {-Return the topic number currently loaded into the help buffers}
(2) 4-389     procedure hwOptionsOn(OptionFlags : Word);
                {-Activate multiple options}
(2) 4-389     procedure hwOptionsOff(OptionFlags : Word);
                {-Deactivate multiple options}
(2) 4-389     function hwOptionsAreOn(OptionFlags : Word) : Boolean;
                {-Return True if all specified options are on}
(2) 4-401     procedure SetPickPadSize(PadSize : Byte);
                {-Set number of characters for padding of pick items}
(2) 4-400     procedure SetHelpPadSize(Pad : Byte);
                {-Set number of blank characters at edges of help window}
(2) 4-384     procedure AddTopicHeader(PadSize : Byte; MaxLen : Byte;
                                       Posn : HeaderPosType);
                {-Add a header to display the topic name}
```

```
(2) 4-384   procedure AddMoreHelpHeader(PromptStr : String80;
                                        Posn : HeaderPosType;
                                        UpStr, DnStr, SepStr : String10;
                                        UpPos, DnPos, SepPos : Byte);
              {-Add a specialized more header to indicate more help}
(2) 4-394   procedure InitHelpFrame;
              {-Make help frame look like wFrame}
(2) 4-401   procedure SetPrevTopicHotSpot(SpotNum : Byte; Posn : FrameCharType);
              {-Activate a hot spot for previous topic command}
(2) 4-400   procedure SetNestedFunc(NF : hwNestedFunc);
              {-Set function to call for nested xref commands}
(2) 4-388   function HighestTopic : Word;
              {-Return highest topic number in help system}
(2) 4-398   function PickTopics : Word;
              {-Return number of help index topics in help system}
(2) 4-387   function FindTopic(Name : String;
                               MatchFunc : hwMatchFunc) : Word; virtual;
              {-Return Topic number of help with specified Name, 0 if not found}
(2) 4-399   function ScreenWordXY(XPos, YPos : Byte;
                                  ScanBack, ScanForward : Boolean) : String;
              {-Parse word from screen at absolute (XPos, YPos)}
(2) 4-385   function CurrentXrefTopic : Word; virtual;
              {-Return help topic of current cross reference, $FFFF if none}
(2) 4-397   constructor Load(var S : IdStream);
              {-Load a help system from a stream}
(2) 4-403   procedure Store(var S : IdStream);
              {-Store a help system in a stream}
```

OpHelp ScrollingHelpWindow

```
(2) 4-416   procedure SetMaxScroll(MaxFirstCol : Word);
              {-Set a new value for maximum horizontal scrolling}
(2) 4-416   procedure SetSendFunc(SF : shSendFunc);
              {-Set function to be called by ccBlkCopy command}
(2) 4-415   procedure SetBlockAttr(Color, Mono : Byte);
              {-Set the attribute for blocks}
(2) 4-415   procedure ResetBlock;
              {-Hide and clear the current block. No screen update}
(2) 4-415   procedure HideBlock;
              {-Hide the current block. No screen update}
```

OpHelp Procedures and Functions

```
(2) 4-418   function ScrollingHelpFile(HelpFile : PathStr) : Boolean;
              {-Return True if HelpFile is designed for a scrolling help window}
(2) 4-387   function DefaultTopicMatch(VaryingName, FixedName : String;
                                       HPtr : AbstractHelpWindowPtr) : Boolean;
              {-Default match function}
(2) 4-379   function NoNestedAction(HPtr : AbstractHelpWindowPtr) : Boolean;
              {-Defaulted nested xref function that does nothing}
```

OpMemo Memo

```
(2) 5-10    constructor Init(X1, Y1, X2, Y2 : Byte; BufferSize : Word;
                             BufPtr : Pointer);
              {-Create a memo window}
(2) 5-11    constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                                   Options : LongInt; BufferSize : Word;
                                   BufPtr : Pointer);
              {-Create a memo window with custom window options}
(2) 5-10    destructor Done; virtual;
              {-Deallocate text buffer if appropriate}
(2) 5-14    procedure Process; virtual;
              {-Process editing commands}
```

```
(2) 5-14    procedure ReinitBuffer;
              {-Reinitialize internal variables when contents of buffer have
                changed}
(2) 5-13    procedure meOptionsOn(OptionFlags : LongInt);
              {-Turn options on}
(2) 5-13    procedure meOptionsOff(OptionFlags : LongInt);
              {-Turn options off}
(2) 5-13    function meOptionsAreOn(OptionFlags : LongInt) : Boolean;
              {-Return true if all specified options are on}
(2) 5-15    procedure SetCtrlAttr(Color, Mono : Byte);
              {-Set attributes for control characters}
(2) 5-17    procedure SetStatusProc(MSP : MemoStatusProc);
              {-Set status procedure}
(2) 5-15    procedure SetLineLimit(N : Integer);
              {-Set limit on number of lines}
(2) 5-17    procedure SetTabSize(TabSize : Byte);
              {-Set size for fixed tabs}
(2) 5-16    procedure SetRightMargin(RM : Byte);
              {-Set right margin}
(2) 5-16    procedure SetMaxLength(Len : Byte);
              {-Set maximum line length}
(2) 5-12    constructor Load(var S : IdStream);
              {-Load a memo from a stream}
(2) 5-18    procedure Store(var S : IdStream);
              {-Store a memo in a stream}
```

OpMemo MemoFile

```
(2) 5-22    constructor Init(X1, Y1, X2, Y2 : Byte; BufferSize : Word;
                             BufPtr : Pointer);
              {-Create a memo window for editing files}
(2) 5-23    constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                                   Options : LongInt; BufferSize : Word;
                                   BufPtr : Pointer);
              {-Create a memo window for editing files}
(2) 5-22    constructor InitAndAlloc(X1, Y1, X2, Y2 : Byte; BufferSize : Word);
              {-Create a memo window for editing files and allocate a buffer}
(2) 5-24    constructor InitCustomAndAlloc(X1, Y1, X2, Y2 : Byte;
                                           var Colors : ColorSet;
                                           Options : LongInt; BufferSize : Word);
              {-Create a memo window for editing files and allocate a buffer}
(2) 5-26    procedure ReadFile(FName : string; var FSize : LongInt);
              {-Read a file into the buffer, returning a status code}
(2) 5-26    procedure SaveFile;
              {-Save the text buffer in mfFileName}
(2) 5-25    constructor Load(var S : IdStream);
              {-Load a memofile from a stream}
(2) 5-27    procedure Store(var S : IdStream);
              {-Store a memofile in a stream}
```

OpMemo Procedures and Functions

```
(2) 5-29    procedure MemoStatus(MP : MemoPtr);
              {-Display status line}
(2) 5-29    procedure MemoError(UnitCode : Byte; var ErrorCode : Word;
                                ErrorMsg : string);
              {-Display error message and wait for key press}
```

OpEditor TextEditor

```
(2) 5-42    constructor Init(X1, Y1, X2, Y2 : Byte; BufferSize : Word);
              {-Create a window for editing files}
(2) 5-44    constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                                   Options : LongInt; BufferSize : Word);
              {-Create a custom window for editing files}
```

```
(2) 5-42    constructor InitBuf(X1, Y1, X2, Y2 : Byte; BufferSize : Word;
                                BufPtr : Pointer);
              {-Create a window for editing files, buffer already allocated}
(2) 5-42    constructor InitBufCustom(X1, Y1, X2, Y2 : Byte;
                                      var Colors : ColorSet;
                                      Options : LongInt; BufferSize : Word;
                                      BufPtr : Pointer);
              {-Create a custom window for editing files, buffer already
                allocated}
(2) 5-46    procedure Process; virtual;
              {-Process editing commands}
(2) 5-55    procedure teOptionsOn(OptionFlags : LongInt);
              {-Turn options on}
(2) 5-55    procedure teOptionsOff(OptionFlags : LongInt);
              {-Turn options off}
(2) 5-55    function teOptionsAreOn(OptionFlags : LongInt) : Boolean;
              {-Return true if all specified options are on}
(2) 5-48    procedure SetEditProc(EF : EditorEditFunc);
              {-Set edit function}
(2) 5-49    procedure SetGetFileProc(GFF : EditorGetFileFunc);
              {-Set get file function}
(2) 5-52    procedure SetYesNoProc(YNF : EditorYesNoFunc);
              {-Set yes-no function}
(2) 5-47    procedure SetBlockAttr(Color, Mono : Byte);
              {-Set attributes for marked blocks}
(2) 5-51    procedure SetMarkerAttr(Color, Mono : Byte);
              {-Set attributes for text markers}
(2) 5-51    procedure SetHighlightAttr(Color, Mono : Byte);
              {-Set attributes for highlighted (found) text}
(2) 5-47    procedure SetBlockIndent(Indent : Byte);
              {-Set block indentation level}
(2) 5-47    procedure SetDefaultExtension(DefExt : ExtStr);
              {-Default extension to use when prompting for filenames}
(2) 5-51    procedure SetPrinter(LptNum : Byte);
              {-Set printer (1-3)}
(2) 5-41    procedure GotoLineCol(LineNum : Integer; Col : Byte);
              {-Move cursor to LineNum, Col. Must not be called until editor
                window has been displayed once.}
(2) 5-41    procedure GotoOffset(Offset : Word);
              {-Move cursor to specified offset in the text buffer. Must not be
                called until editor window has been displayed once.}
(2) 5-45    procedure InsertTextAtCursor(var Data; Bytes : Word);
              {-Insert specified number of Bytes of Data into the text buffer at
                the cursor}
(2) 5-45    procedure InsertStringAtCursor(S : String);
              {-Insert string S into the text buffer at the cursor}
(2) 5-46    constructor Load(var S : IdStream);
              {-Load a text editor from a stream}
(2) 5-54    procedure Store(var S : IdStream);
            {-Store a text editor in a stream}
```

OpBrowse Browser

```
(2) 5-67    constructor Init(X1, Y1, X2, Y2 : Byte; HeapToUse : LongInt);
              {-Initialize the browser}
(2) 5-67    constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                                   Options : LongInt; HeapToUse : LongInt);
              {-Initialize the browser with custom window options}
(2) 5-65    destructor Done; virtual;
              {-Deallocate page buffers}
(2) 5-69    procedure Process; virtual;
              {-Process browse commands}
(2) 5-68    procedure OpenFile(FName : String);
              {-Open file for browsing}
```

```
(2) 5-65    procedure CloseFile;
              {-Close the current browse file}
(2) 5-65    function CurrentFileName : String;
              {-Return the name of the current browse file}
(2) 5-64    procedure brOptionsOn(OptionFlags : Word);
              {-Activate multiple options}
(2) 5-64    procedure brOptionsOff(OptionFlags : Word);
              {-Deactivate multiple options}
(2) 5-64    function brOptionsAreOn(OptionFlags : Word) : Boolean;
              {-Return true if all specified options are on}
(2) 5-76    procedure ToggleModes;
              {-Toggle between hex and ASCII modes}
(2) 5-76    procedure ToggleHighBitStripping;
              {-Toggle stripping of high bits}
(2) 5-69    procedure SetDefaultExtension(DefExt : ExtStr);
              {-Default extension to use when prompting for filenames}
(2) 5-73    procedure SetPrinter(LptNum : Byte);
              {-Set printer (1-3)}
(2) 5-74    procedure SetStatusProc(SP : BrowseStatusProc);
              {-Set status procedure}
(2) 5-70    procedure SetEditProc(EF : BrowseEditFunc);
              {-Set edit function}
(2) 5-71    procedure SetGetFileProc(GF : BrowseGetFileFunc);
              {-Set edit function}
(2) 5-69    procedure SetBlockAttr(Color, Mono : Byte);
              {-Set attributes for marked blocks}
(2) 5-73    procedure SetMarkerAttr(Color, Mono : Byte);
              {-Set attributes for text markers}
(2) 5-73    procedure SetHighlightAttr(Color, Mono : Byte);
              {-Set attributes for found text}
(2) 5-68    constructor Load(var S : IdStream);
              {-Load a browser from a stream}
(2) 5-75    procedure Store(var S : IdStream);
              {-Store a browser in a stream}
(2) 5-66    procedure GotoLine(LineNum : Word);
              {-Scroll display to the specified line}
(2) 5-66    procedure GotoOffset(Offset : LongInt);
              {-Scroll display to the line nearest Offset}
(2) 5-76    procedure Transfer(StartPos, StopPos : LongInt;
                               var Buffer; BufSize : Word);
              {-Copy browse characters from StartPos to StopPos into Buffer}
```

OpBrowse Procedures and Functions

```
(2) 5-78    procedure BrowseStatus(BP : BrowserPtr);
              {-Display status line}
```

OpAbsFld

```
(2) 6-13    procedure ChangeDecimalChar(DC : Char);
              {-Reset DecimalChar to DC}
(2) 6-13    function InternationalCurrency(FormChar : Char; MaxDigits : Byte;
                                           FloatIfPossible : Boolean;
                                           AddCommas : Boolean) : string;
              {-Return a picture mask for a currency string, based on DOS's
                country info}
```

OpField SelectField

```
(2) 6-18    procedure sfOptionsOn(OptionFlags : LongInt);
              {-Activate multiple options}
(2) 6-18    procedure sfOptionsOff(OptionFlags : LongInt);
              {-Deactivate multiple options}
(2) 6-18    function sfOptionsAreOn(OptionFlags : LongInt) : Boolean;
              {-Return true if all specified options are on}
```

```
(2) 6-18    procedure SetPromptAttr(Color, Mono : Byte);
              {-Set attributes for prompts}
(2) 6-18    procedure SetSelectedPromptAttr(Color, Mono : Byte);
              {-Set attributes for selected prompts}
(2) 6-18    procedure SetProtectedPromptAttr(Color, Mono : Byte);
              {-Set attributes for protected prompts}
(2) 6-18    procedure SetFieldAttr(Color, Mono : Byte);
              {-Set attributes for unselected fields}
(2) 6-18    procedure SetSelectedFieldAttr(Color, Mono : Byte);
              {-Set attributes for selected fields}
(2) 6-18    procedure SetProtectedFieldAttr(Color, Mono : Byte);
              {-Set attributes for protected fields}
(2) 6-18    procedure SetCtrlAttr(Color, Mono : Byte);
              {-Set attributes for control characters}
```

OpField EntryField

```
(2) 6-26    procedure efOptionsOn(OptionFlags : LongInt);
              {-Activate multiple options}
(2) 6-26    procedure efOptionsOff(OptionFlags : LongInt);
              {-Deactivate multiple options}
(2) 6-26    function efOptionsAreOn(OptionFlags : LongInt) : Boolean;
              {-Return true if all specified options are on}
(2) 6-26    procedure sefOptionsOn(OptionFlags : LongInt);
              {-Activate multiple secondary options}
(2) 6-26    procedure sefOptionsOff(OptionFlags : LongInt);
              {-Deactivate multiple secondary options}
(2) 6-26    function sefOptionsAreOn(OptionFlags : LongInt) : Boolean;
              {-Return true if all specified secondary options are on}
```

OpField Procedures and Functions

```
(2) 6-46    function ValidateNoBlanks(EFP : EntryFieldPtr; var ErrCode : Word;
                                      var ErrorSt : StringPtr) : Boolean;
            {-Validate that no usable subfields in a string contain spaces}
(2) 6-46    function ValidateNotPartial(EFP : EntryFieldPtr; var ErrCode : Word;
                                        var ErrorSt : StringPtr) : Boolean;
            {-Validate that no usable subfields in a string contain spaces
              unless all do}
(2) 6-47    function ValidateSubfields(SubfieldMask : string;
                                       EFP : EntryFieldPtr; var ErrCode : Word;
                                       var ErrorSt : StringPtr) : Boolean;
            {-Validate that subfields in a string meet the requirements of
              SubfieldMask. Note: This is NOT a regular validation routine.}
```

OpSelect AbstractSelector

```
(2) 6-72    constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                                   Options : LongInt; var CP : CommandProcessor;
                                   UnitCode : Byte);
              {-Initialize an abstract selector with custom window options}
(2) 6-70    destructor Done; virtual;
              {-Deallocate field list, screen buffers}
(2) 6-70    procedure DrawField(ID : Word);
              {-Draw the specified field}
(2) 6-73    procedure ResetScreen;
              {-Call when contents of field(s) changed without calling DrawField}
(2) 6-80    procedure SetWrapMode(WM : WrapMode);
              {-Set the wrap mode for the selector}
(2) 6-78    procedure SetScreenUpdateProc(SUP : ScreenUpdateProc);
              {-Set procedure to be called after a complete screen update}
(2) 6-77    procedure SetPromptAttr(Color, Mono : Byte);
              {-Set attributes for prompts}
(2) 6-79    procedure SetSelectedPromptAttr(Color, Mono : Byte);
              {-Set attributes for selected prompts}
```

(2) 6-78
```
procedure SetProtectedPromptAttr(Color, Mono : Byte);
  {-Set attributes for protected prompts}
```
(2) 6-75
```
procedure SetFieldAttr(Color, Mono : Byte);
  {-Set attributes for unselected fields}
```
(2) 6-79
```
procedure SetSelectedFieldAttr(Color, Mono : Byte);
  {-Set attributes for selected fields}
```
(2) 6-77
```
procedure SetProtectedFieldAttr(Color, Mono : Byte);
  {-Set attributes for protected fields}
```
(2) 6-75
```
procedure SetCtrlAttr(Color, Mono : Byte);
  {-Set attributes for control characters}
```
(2) 6-67
```
procedure ChangePromptAttr(ID : Word; Color, Mono : Byte);
  {-Change attributes for prompts}
```
(2) 6-69
```
procedure ChangeSelectedPromptAttr(ID : Word; Color, Mono : Byte);
  {-Change attributes for selected prompts}
```
(2) 6-68
```
procedure ChangeProtectedPromptAttr(ID : Word; Color, Mono : Byte);
  {-Change attributes for protected prompts}
```
(2) 6-66
```
procedure ChangeFieldAttr(ID : Word; Color, Mono : Byte);
  {-Change attributes for unselected fields}
```
(2) 6-69
```
procedure ChangeSelectedFieldAttr(ID : Word; Color, Mono : Byte);
  {-Change attributes for selected fields}
```
(2) 6-67
```
procedure ChangeProtectedFieldAttr(ID : Word; Color, Mono : Byte);
  {-Change attributes for protected fields}
```
(2) 6-65
```
procedure ChangeCtrlAttr(ID : Word; Color, Mono : Byte);
  {-Change attributes for control characters}
```
(2) 6-64
```
procedure AddTextField(St : string; Row, Col : Word);
  {-Add a text field}
```
(2) 6-64
```
procedure AddTextFieldCustom(St : string; Row, Col : Word;
                             Color, Mono : Byte);
  {-Add a text field with specific colors}
```
(2) 6-65
```
procedure AddTextFieldDeluxe(St : string; Row, Col : Word;
                             var ColorAttrs, MonoAttrs : FlexAttrs);
  {-Add a text field with special colors}
```
(2) 6-70
```
function FindField(ID : Word) : SelectFieldPtr;
  {-Return a pointer to the specified field}
```
(2) 6-71
```
function FindTextField(ID : Word) : TextFieldPtr;
  {-Return a pointer to the specified text field}
```
(2) 6-76
```
procedure SetNextField(ID : Word);
  {-Set value of asNext}
```
(2) 6-71
```
function GetCurrentID : Word;
  {-Get the ID for the current field}
```
(2) 6-72
```
function GetHelpIndex(ID : Word) : Word;
  {-Get the help index for the specified field}
```
(2) 6-74
```
procedure SetAllFieldLinks(var LinksMap);
  {-Set the forward and backward links for all fields}
```
(2) 6-76
```
procedure SetFieldLinks(ID, Next, Prev : Word);
  {-Specify the fields to jump to when <Tab> or <ShTab> pressed on a
    given field}
```
(2) 6-68
```
procedure ChangeProtection(ID : Word; On : Boolean);
  {-Change the protection status of the specified field}
```
(2) 6-66
```
procedure ChangeHidden(ID : Word; On : Boolean);
  {-Change the hidden status of the specified field}
```
(2) 6-66
```
procedure ChangeInvisibility(ID : Word; On : Boolean);
  {-Change the invisibility status of the specified field}
```
(2) 6-73
```
procedure PositionCursorAt(SFP : SelectFieldPtr); virtual;
  {-Puts the cursor on the beginning of the specified field}
```
(2) 6-80
```
procedure VisitAllSelectFields(FProc : asFieldProc; var D);
  {-Call the specified procedure for all fields in the selector}
```
(2) 6-80
```
procedure VisitAllTextFields(FProc : asTextFieldProc; var D);
  {-Call the specified procedure for all text fields in the selector}
```

OpSelect Selector

(2) 6-91
```
constructor Init(X1, Y1, X2, Y2 : Byte);
  {-Initialize the selector}
```

OPSELECT

```
(2) 6-91    constructor InitCustom(X1, Y1, X2, Y2 : Byte;
                                   var Colors : ColorSet;
                                   Options : LongInt);
              {-Initialize a selector with custom window options}
(2) 6-93    procedure Process; virtual;
              {-Process selection commands}
(2) 6-90    procedure AddField(PromptRow, PromptCol : Word; PromptWidth : Byte;
                               FieldRow, FieldCol : Word; FieldWidth : Byte;
                               HelpIndex : Word);
              {-Add a field with the specified coordinates and help index}
(2) 6-97    procedure slOptionsOn(OptionFlags : Byte);
              {-Activate multiple options}
(2) 6-97    procedure slOptionsOff(OptionFlags : Byte);
              {-Deactivate multiple options}
(2) 6-97    function slOptionsAreOn(OptionFlags : Byte) : Boolean;
              {-Return true if all specified options are on}
(2) 6-96    procedure slFieldOptionsOn(OptionFlags : LongInt);
              {-Activate multiple options}
(2) 6-96    procedure slFieldOptionsOff(OptionFlags : LongInt);
              {-Deactivate multiple options}
(2) 6-95    function slFieldOptionsAreOn(OptionFlags : LongInt) : Boolean;
              {-Return true if all specified options are on}
(2) 6-94    procedure SetGetFieldProc(GFP : GetFieldProc);
              {-Set routine to call to get strings to display}
(2) 6-93    procedure SetActionProc(AP : SelectProc);
              {-Set routine to call when field is selected}
(2) 6-95    procedure SetPreSelectProc(PSP : SelectProc);
              {-Set routine to call before moving to new field}
(2) 6-94    procedure SetPostSelectProc(PSP : SelectProc);
              {-Set routine to call before moving to next one}
(2) 6-92    constructor Load(var S : IdStream);
              {-Load a selector from a stream}
(2) 6-98    procedure Store(var S : IdStream);
              {-Store a selector in a stream}
```

OpSelect ScrollingSelector

```
(2) 6-104   constructor Init(X1, Y1, X2, Y2 : Byte);
              {-Initialize the selector}
(2) 6-104   constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                                   Options : LongInt);
              {-Initialize a selector with custom window options}
(2) 6-103   destructor Done; virtual;
              {-Deallocate field list, screen buffers}
(2) 6-103   procedure AllocateScreen;
              {-Call after last field added to allocate virtual screen}
(2) 6-108   procedure VScreenToScreen; virtual;
              {-Copy the virtual screen to the physical screen}
(2) 6-105   constructor Load(var S : IdStream);
              {-Load a selector from a stream}
(2) 6-106   procedure Store(var S : IdStream);
              {-Store a selector in a stream}
```

OpEntry EntryScreen

```
(2) 6-153   constructor Init(X1, Y1, X2, Y2 : Byte);
              {-Initialize the entry screen}
(2) 6-153   constructor InitCustom(X1, Y1, X2, Y2 : Byte;
                                   var Colors : ColorSet;
                                   Options : LongInt);
              {-Initialize the entry screen with custom window options}
(2) 6-147   destructor Done; virtual;
              {-Dispose of the entry screen}
(2) 6-155   procedure Process; virtual;
              {-Process data entry commands}
```

```
(2) 6-116     procedure AddStringField(Prompt : string;  pRow, pCol : Word;
                                       Picture : string; fRow, fCol : Word;
                                       fWidth : Byte;    HelpIndex : Word;
                                       var EditSt : string);
                {-Add a field of type string}
(2) 6-118     procedure AddSimpleStringField(Prompt : string; pRow, pCol : Word;
                                             PicChar : Char;  fRow, fCol : Word;
                                             fWidth, MaxLen : Byte;
                                             HelpIndex : Word;
                                             var EditSt : string);
                {-Add a field of type string (simple)}
(2) 6-116     procedure AddArrayField(Prompt : string;  pRow, pCol : Word;
                                      Picture : string; fRow, fCol : Word;
                                      fWidth : Byte;    HelpIndex : Word;
                                      var EditVar);
                {-Add a field of type array (of char)}
(2) 6-119     procedure AddCharField(Prompt : string;  pRow, pCol : Word;
                                     Picture : string; fRow, fCol : Word;
                                     HelpIndex : Word; CharLo, CharHi : Char;
                                     var EditChar : Char);
                {-Add a field of type char}
(2) 6-120     procedure AddSimpleCharField(Prompt : string;    pRow, pCol : Word;
                                           PicChar : Char;     fRow, fCol : Word;
                                           HelpIndex : Word;
                                           CharLo, CharHi : Char;
                                           var EditChar : Char);
                {-Add a field of type char (simple)}
(2) 6-120     procedure AddBooleanField(Prompt : string;  pRow, pCol : Word;
                                        Picture : string; fRow, fCol : Word;
                                        HelpIndex : Word;
                                        var EditBool : Boolean);
                {-Add a field of type boolean}
(2) 6-120     procedure AddSimpleBooleanField(Prompt : string; pRow, pCol : Word;
                                              fRow, fCol : Word;
                                              HelpIndex : Word;
                                              var EditBool : Boolean);
                {-Add a field of type boolean (simple)}
(2) 6-120     procedure AddYesNoField(Prompt : string;  pRow, pCol : Word;
                                      Picture : string; fRow, fCol : Word;
                                      HelpIndex : Word; var EditYesNo : Boolean);
                {-Add a field of type yes/no}
(2) 6-120     procedure AddSimpleYesNoField(Prompt : string;   pRow, pCol : Word;
                                            fRow, fCol : Word; HelpIndex : Word;
                                            var EditYesNo : Boolean);
                {-Add a field of type yes/no (simple)}
(2) 6-116     procedure AddLongField(Prompt : string;  pRow, pCol : Word;
                                     Picture : string; fRow, fCol : Word;
                                     HelpIndex : Word; LongLo, LongHi : LongInt;
                                     var EditLong : LongInt);
                {-Add a field of type LongInt}
(2) 6-120     procedure AddNumericLongField(...);
                {-Add a field of type LongInt (numeric)}
(2) 6-118     procedure AddSimpleLongField(Prompt : string; pRow, pCol : Word;
                                           PicChar : Char;  fRow, fCol : Word;
                                           fWidth : Byte;   HelpIndex : Word;
                                           LongLo, LongHi : LongInt;
                                           var EditLong : LongInt);
                {-Add a field of type LongInt (simple)}
(2) 6-116     procedure AddWordField(Prompt : string;  pRow, pCol : Word;
                                     Picture : string; fRow, fCol : Word;
                                     HelpIndex : Word; WordLo, WordHi : Word;
                                     var EditWord : Word);
                {-Add a field of type word}
(2) 6-120     procedure AddNumericWordField(...);
                {-Add a field of type word (numeric)}
```

```
(2) 6-118     procedure AddSimpleWordField(Prompt : string;    pRow, pCol : Word;
                                           PicChar : Char; fRow, fCol : Word;
                                           fWidth : Byte;      HelpIndex : Word;
                                           WordLo, WordHi : Word;
                                           var EditWord : Word);
                {-Add a field of type word (simple)}
(2) 6-116     procedure AddIntField(Prompt : string;  pRow, pCol : Word;
                                    Picture : string; fRow, fCol : Word;
                                    HelpIndex : Word; IntLo, IntHi : Integer;
                                    var EditInt : Integer);
                {-Add a field of type integer}
(2) 6-121     procedure AddNumericIntField(...);
                {-Add a field of type integer (numeric)}
(2) 6-118     procedure AddSimpleIntField(Prompt : string;    pRow, pCol : Word;
                                          PicChar : Char;     fRow, fCol : Word;
                                          fWidth : Byte;      HelpIndex : Word;
                                          IntLo, IntHi : Integer;
                                          var EditInt : Integer);
                {-Add a field of type integer (simple)}
(2) 6-116     procedure AddByteField(Prompt : string;  pRow, pCol : Word;
                                     Picture : string; fRow, fCol : Word;
                                     HelpIndex : Word; ByteLo, ByteHi : Byte;
                                     var EditByte : Byte);
                {-Add a field of type byte}
(2) 6-121     procedure AddNumericByteField(...);
                {-Add a field of type byte (numeric)}
(2) 6-118     procedure AddSimpleByteField(Prompt : string;    pRow, pCol : Word;
                                           PicChar : Char;     fRow, fCol : Word;
                                           fWidth : Byte;      HelpIndex : Word;
                                           ByteLo, ByteHi : Byte;
                                           var EditByte : Byte);
                {-Add a field of type byte (simple)}
(2) 6-116     procedure AddShortField(Prompt : string;   pRow, pCol : Word;
                                      Picture : string;  fRow, fCol : Word;
                                      HelpIndex : Word;
                                      ShortLo, ShortHi : ShortInt;
                                      var EditShort : ShortInt);
                {-Add a field of type shortint}
(2) 6-121     procedure AddNumericShortField(...);
                {-Add a field of type shortint (numeric)}
(2) 6-118     procedure AddSimpleShortField(Prompt : string; pRow, pCol : Word;
                                            PicChar : Char;  fRow, fCol : Word;
                                            fWidth : Byte;   HelpIndex : Word;
                                            ShortLo, ShortHi : ShortInt;
                                            var EditShort : ShortInt);
                {-Add a field of type shortint (simple)}
(2) 6-116     procedure AddRealField(Prompt : string;   pRow, pCol : Word;
                                     Picture : string;  fRow, fCol : Word;
                                     HelpIndex : Word;  RealLo, RealHi : Real;
                                     Places : Byte;     var EditReal : Real);
                {-Add a field of type real}
(2) 6-121     procedure AddNumericRealField(...);
                {-Add a field of type real (numeric)}
(2) 6-118     procedure AddSimpleRealField(Prompt : string; pRow, pCol : Word;
                                           PicChar : Char;  fRow, fCol : Word;
                                           fWidth : Byte;   HelpIndex : Word;
                                           RealLo, RealHi : Real; Places : Byte;
                                           var EditReal : Real);
                {-Add a field of type real (simple)}
(2) 6-117     procedure AddDateField(Prompt : string;      pRow, pCol : Word;
                                     Picture : DateString; fRow, fCol : Word;
                                     HelpIndex : Word;     DateLo, DateHi : Date;
                                     var EditDate : Date);
                {-Add a field of type date}
```

```
(2) 6-117    procedure AddDateStField(Prompt : string;      pRow, pCol : Word;
                                      Picture : DateString; fRow, fCol : Word;
                                      HelpIndex : Word;
                                      var EditDate : DateString);
               {-Add a field of type date string}
(2) 6-117    procedure AddTimeField(Prompt : string;      pRow, pCol : Word;
                                    Picture : DateString; fRow, fCol : Word;
                                    HelpIndex : Word;     TimeLo, TimeHi : Time;
                                    var EditTime : Time);
               {-Add a field of type time}
(2) 6-117    procedure AddNestedField(Prompt : string;  pRow, pCol : Word;
                                      Picture : string; fRow, fCol : Word;
                                      fWidth : Byte;    HelpIndex : Word);
               {-Add a field that is associated with a nested form}
(2) 6-118    procedure AddNestedStringField(Prompt : string;  pRow, pCol : Word;
                                            Picture : string; fRow, fCol : Word;
                                            fWidth : Byte;    HelpIndex : Word;
                                            var EditSt : string);
               {-Add a string field that is associated with a nested form}
(2) 6-122    procedure AddChoiceField(Prompt : string;  pRow, pCol : Word;
                                      Picture : string; fRow, fCol : Word;
                                      HelpIndex : Word; DataSize : Byte;
                                      Increment : IncChoiceProc;
                                      var EditVar);
               {-Add a multiple choice field}
(2) 6-122    procedure AddMultiLineField(Prompt : string; pRow, pCol : Word;
                                         PicChar : Char;  fRow, fCol : Word;
                                         fWidth, fHeight : Byte;
                                         HelpIndex : Word; var EditVar);
               {-Add a multi-line field}
(2) 6-122    procedure AddPickStringField(Prompt : string; pRow, pCol : Word;
                                          fRow, fCol : Word; fWidth : Byte;
                                          HelpIndex : Word;
                                          var EditSt : string;
                                          var PL : PickList);
               {-Add a field of type string to be edited with a pick list}
(2) 6-122    procedure AddUserField(Prompt : string; pRow, pCol : Word;
                                    Picture : string; fRow, fCol : Word;
                                    fWidth : Byte; HelpIndex : Word;
                                    RangeLo : RangeType; RangeHi : RangeType;
                                    DataSize : Word;     DecimalPlaces : Byte;
                                    Validate: ValidationFunc;
                                    Convert : ConversionProc;
                                    FDraw   : DrawProc;
                                    Editor  : EditProc;
                                    var EditVar);
               {-Add a field of a user-defined type}
(2) 6-160    procedure SetPreEditProc(PEP : esUserProc);
               {-Specify routine to call just before field is edited}
(2) 6-159    procedure SetPostEditProc(PEP : esUserProc);
               {-Specify routine to call after each edit}
(2) 6-162    procedure SetValidateFunc(UVF : esUserValidationFunc);
               {-Specify routine to call before exiting}
(2) 6-148    procedure esOptionsOn(OptionFlags : Byte);
               {-Activate multiple options}
(2) 6-147    procedure esOptionsOff(OptionFlags : Byte);
               {-Deactivate multiple options}
(2) 6-147    function esOptionsAreOn(OptionFlags : Byte) : Boolean;
               {-Return true if all specified options are on}
(2) 6-147    procedure esFieldOptionsOn(OptionFlags : LongInt);
               {-Activate multiple field options}
(2) 6-146    procedure esFieldOptionsOff(OptionFlags : LongInt);
               {-Deactivate multiple field options}
(2) 6-146    function esFieldOptionsAreOn(OptionFlags : LongInt) : Boolean;
               {-Return true if all specified field options are on}
(2) 6-149    procedure esSecFieldOptionsOn(OptionFlags : LongInt);
               {-Activate multiple secondary field options}
```

(2) 6-149
```
procedure esSecFieldOptionsOff(OptionFlags : LongInt);
  {-Deactivate multiple secondary field options}
```
(2) 6-148
```
function esSecFieldOptionsAreOn(OptionFlags : LongInt) : Boolean;
  {-Return true if all specified secondary field options are on}
```
(2) 6-158
```
procedure SetForceMode(Force, Overtype : Boolean);
  {-Force insert or overtype mode, else use previous setting}
```
(2) 6-158
```
procedure SetPadChar(Ch : Char);
  {-Set character used to pad ends of strings}
```
(2) 6-159
```
procedure SetPasswordChar(Ch : Char);
  {-Set character used in password mode}
```
(2) 6-156
```
procedure SetBeepOnError(On : Boolean);
  {-Activate/deactivate beeping on error option for all fields}
```
(2) 6-157
```
procedure SetDelimiters(Left, Right : Char);
  {-Set field delimiters for the entry screen (null = none)}
```
(2) 6-157
```
procedure SetDelimAttr(Color, Mono : Byte);
  {-Set attributes for delimiters (unselected)}
```
(2) 6-161
```
procedure SetSelectedDelimAttr(Color, Mono : Byte);
  {-Set attributes for delimiters (selected)}
```
(2) 6-160
```
procedure SetProtectedDelimAttr(Color, Mono : Byte);
  {-Set attributes for delimiters (protected)}
```
(2) 6-154
```
constructor Load(var S : IdStream);
  {-Load an entry screen from a stream}
```
(2) 6-163
```
procedure Store(var S : IdStream);
  {-Store an entry screen in a stream}
```
(2) 6-161
```
procedure SetUserRecord(var UserRec; UserRecSize : Word);
  {-Set the address and size of the user record}
```
(2) 6-152
```
function GetUserRecord : Pointer;
  {-Return a pointer to the user record}
```
(2) 6-152
```
function GetUserRecordSize : Word;
  {-Return the size of the user record}
```
(2) 6-144
```
procedure ChangeRequired(ID : Word; On : Boolean);
  {-Modify the required status of a field after it has been added}
```
(2) 6-145
```
procedure ChangeValidation(ID : Word; VF : ValidationFunc);
  {-Change the validation routine for the specified field}
```
(2) 6-144
```
procedure ChangeConversion(ID : Word; CP : ConversionProc);
  {-Change the conversion routine for the specified field}
```
(2) 6-144
```
procedure ChangeRange(ID : Word; var RangeLo, RangeHi);
  {-Change the valid range for the specified field}
```
(2) 6-145
```
function CurrentFieldModified : Boolean;
  {-Return True if current field was modified. Call only from
    within a post-edit routine.}
```
(2) 6-150
```
function EvaluateCommand(var Cmd : Word) : Word;
  {-Given a command, return the ID for the field the cursor will
    move to next. Cmd may be modified on return. This routine is
    intended to be called only from within a post-edit routine!}
```
(2) 6-151
```
procedure GetFieldCoordinates(var Row, Col, Wid : Byte);
  {-Returns the absolute screen coordinates for the current field.
    This routine is intended to be called only from within a
    pre-edit routine!}
```
(2) 6-151
```
function GetLastField : EntryFieldPtr;
  {-Return pointer to last field in entry screen}
```
(2) 6-156
```
function RevalidateAll(var EFP : EntryFieldPtr; var ErrCode : Word;
                       var ErrMsg : StringPtr) : Boolean;
  {-Re-validate all unprotected fields in entry screen. If an error
    is found, EFP will point to the field with the error, an error
    code and error message will be returned, and RevalidateAll will
    be False.}
```
(2) 6-165
```
procedure VisitAllEntryFields(FProc : esFieldProc; var D);
  {-Call the specified procedure for all fields in the entry
    screen}
```

OpEntry ScrollingEntryScreen

(2) 6-170
```
constructor Init(X1, Y1, X2, Y2 : Byte);
  {-Initialize the entry screen}
```

```
(2) 6-170   constructor InitCustom(X1, Y1, X2, Y2 : Byte;
                                   var Colors : ColorSet; Options : LongInt);
              {-Initialize the entry screen with custom window options}
(2) 6-169   destructor Done; virtual;
              {-Deallocate field list, screen buffers}
(2) 6-169   procedure AllocateScreen;
              {-Call after last field added to allocate virtual screen}
(2) 6-172   procedure VScreenToScreen; virtual;
              {-Copy the virtual screen to the physical screen}
(2) 6-171   constructor Load(var S : IdStream);
              {-Load a scrolling entry screen from a stream}
(2) 6-172   procedure Store(var S : IdStream);
              {-Store a scrolling entry screen in a stream}
```

OpEdit LineEditor

```
(2) 6-189   constructor Init(var Colors : ColorSet);
              {-Sets default values for attributes, etc.}
(2) 6-188   function GetLastCommand : Word;
              {-Return last command entered by user}
(2) 6-188   function GetLastKey : Word;
              {-Return last keystroke entered by user}
(2) 6-188   function GetLastError : Word;
              {-Get last error code}
(2) 6-190   procedure leEditOptionsOn(OptionFlags : LongInt);
              {-Activate multiple options}
(2) 6-189   function leEditOptionsAreOn(OptionFlags : LongInt) : Boolean;
              {-Return true if all specified options are on}
(2) 6-190   procedure leEditOptionsOff(OptionFlags : LongInt);
              {-Deactivate multiple options}
(2) 6-192   procedure leSecEditOptionsOn(OptionFlags : LongInt);
              {-Activate multiple options}
(2) 6-191   function leSecEditOptionsAreOn(OptionFlags : LongInt) : Boolean;
              {-Return true if all specified options are on}
(2) 6-191   procedure leSecEditOptionsOff(OptionFlags : LongInt);
              {-Deactivate multiple options}
(2) 6-195   procedure SetPadChar(Ch : Char);
              {-Set character used to pad strings being edited}
(2) 6-195   procedure SetPasswordChar(Ch : Char);
              {-Set character used to represent text in password mode}
(2) 6-195   procedure SetHelpIndex(Index : Word);
              {-Set the help index}
(2) 6-193   procedure SetCommandProcessor(var CP : CommandProcessor);
              {-Set command processor to use}
(2) 6-194   procedure SetErrorProc(EEP : ErrorProc);
              {-Set procedure to call on error}
(2) 6-192   constructor Load(var S : IdStream);
              {-Load a line editor from a stream}
(2) 6-196   procedure Store(var S : IdStream);
              {-Store a line editor in a stream}
(2) 6-178   procedure ReadString(Prompt : string; Row, Col, MaxLen, Width : Byte;
                                 var St : string);
              {-Edit a string}
(2) 6-178   procedure ReadSpecial(Prompt : string;
                                  Row, Col, MaxLen, Width : Byte;
                                  PictureChar : Char; var St : string);
              {-Edit a string}
(2) 6-179   procedure ReadChar(Prompt : string; Row, Col : Byte;
                               ValidChars : CharSet; var Ch : Char);
              {-Display a prompt and wait for a key in ValidChars}
(2) 6-179   function YesOrNo(Prompt : string; Row, Col : Byte;
                             Default : Char) : Boolean;
              {-Return true for yes, false for no}
```

OPEDIT

```
(2) 6-178     procedure ReadLongInt(Prompt : string; Row, Col, Width : Byte;
                                    LLo, LHi : LongInt; var L : LongInt);
                {-Edit a long integer}
(2) 6-178     procedure ReadInteger(Prompt : string; Row, Col, Width : Byte;
                                    NLo, NHi : Integer; var N : Integer);
                {-Edit an integer}
(2) 6-178     procedure ReadWord(Prompt : string; Row, Col, Width : Byte;
                                 WLo, WHi : Word; var W : Word);
                {-Edit a word}
(2) 6-178     procedure ReadByte(Prompt : string; Row, Col, Width : Byte;
                                 BLo, BHi : Byte; var B : Byte);
                {-Edit a byte}
(2) 6-178     procedure ReadShortInt(Prompt : string; Row, Col, Width : Byte;
                                     SLo, SHi : ShortInt; var SI : Shortint);
                {-Edit a ShortInt}
(2) 6-178     procedure ReadReal(Prompt : string; Row, Col, Width, Places : Byte;
                                 RLo, RHi : Real; var R : Real);
                {-Edit a real}
(2) 6-179     procedure EditString(Prompt : string; Row, Col : Byte;
                                   Picture : string;
                                   Width : Byte; var St : string);
                {-Edit a string}
(2) 6-179     procedure EditArray(Prompt : string; Row, Col : Byte;
                                  Picture : string; Width : Byte; var A);
                {-Edit an array of char}
(2) 6-181     procedure EditChar(Prompt : string; Row, Col : Byte;
                                 Picture : string; ChLo, ChHi : Char;
                                 var Ch : Char);
                {-Edit an array of char}
(2) 6-181     procedure EditBoolean(Prompt : string; Row, Col : Byte;
                                    Picture : string; var B : Boolean);
                {-Edit a boolean variable}
(2) 6-181     procedure EditYesNo(Prompt : string; Row, Col : Byte;
                                  Picture : string; var YN : Boolean);
                {-Edit a yes-no variable}
(2) 6-180     procedure EditLongInt(Prompt : string; Row, Col : Byte;
                                    Picture : string; LLo, LHi : LongInt;
                                    var L : LongInt);
                {-Edit a long integer}
(2) 6-181     procedure EditNumericLongInt(...);
                {-Edit a long integer (numeric)}
(2) 6-180     procedure EditInteger(Prompt : string; Row, Col : Byte;
                                    Picture : string; NLo, NHi : Integer;
                                    var N : Integer);
                {-Edit an integer}
(2) 6-182     procedure EditNumericInteger(...);
                {-Edit an integer (numeric)}
(2) 6-180     procedure EditWord(Prompt : string; Row, Col : Byte;
                                 Picture : string; WLo, WHi : Word;
                                 var W : Word);
                {-Edit a word}
(2) 6-182     procedure EditNumericWord(...);
                {-Edit a word (numeric)}
(2) 6-180     procedure EditByte(Prompt : string; Row, Col : Byte;
                                 Picture : string; BLo, BHi : Byte;
                                 var B : Byte);
                {-Edit a byte}
(2) 6-182     procedure EditNumericByte(...);
                {-Edit a byte (numeric)}
(2) 6-180     procedure EditShortInt(Prompt : string; Row, Col : Byte;
                                     Picture : string; SLo, SHi : ShortInt;
                                     var SI : Shortint);
                {-Edit a ShortInt}
(2) 6-182     procedure EditNumericShortInt(...);
                {-Edit a ShortInt (numeric)}
```

(2) 6-180
```
procedure EditReal(Prompt : string; Row, Col : Byte;
                   Picture : string; Places : Byte;
                   RLo, RHi : Real; var R : Real);
  {-Edit a real}
```
(2) 6-182
```
procedure EditNumericReal(...);
  {-Edit a real (numeric)}
```
(2) 6-181
```
procedure EditDate(Prompt : string; Row, Col : Byte;
                   Picture : string; DLo, DHi : Date; var D : Date);
  {-Edit a date variable}
```
(2) 6-181
```
procedure EditDateSt(Prompt : string; Row, Col : Byte;
                     Picture : DateString; var DS : DateString);
  {-Edit a date string}
```
(2) 6-181
```
procedure EditTime(Prompt : string; Row, Col : Byte;
                   Picture : DateString;
                   TLo, THi : Time; var T : Time);
  {-Edit a time variable}
```
(2) 6-182
```
procedure EditChoice(Prompt : string; Row, Col : Byte;
                     Picture : string; DataSize : Byte;
                     Increment : IncChoiceProc; var MC);
  {-Edit a multiple choice variable}
```
(2) 6-182
```
procedure EditMultiLine(Prompt : string; Row, Col : Byte;
                        Width, Height : Byte; var StArray);
{-Edit a multiline field}
```

OpSEdit SimpleLineEditor

(2) 6-202
```
constructor Init(var Colors : ColorSet);
  {-Sets default values for attributes, etc.}
```
(2) 6-201
```
function GetLastCommand : Word;
  {-Return last command entered by user}
```
(2) 6-201
```
function GetLastKey : Word;
  {-Return last keystroke entered by user}
```
(2) 6-202
```
procedure ReadString(Prompt : string; Row, Col, MaxLen, Width : Byte;
                     var S : string);
  {-Edit a string}
```
(2) 6-203
```
procedure seOptionsOn(OptionFlags : LongInt);
  {-Activate multiple options}
```
(2) 6-203
```
function seOptionsAreOn(OptionFlags : LongInt) : Boolean;
  {-Return true if all specified options are on}
```
(2) 6-203
```
procedure seOptionsOff(OptionFlags : LongInt);
  {-Deactivate multiple options}
```
(2) 6-205
```
procedure SetPromptAttr(Color, Mono : Byte);
  {-Set attributes for prompts}
```
(2) 6-204
```
procedure SetFieldAttr(Color, Mono : Byte);
  {-Set attributes for line being edited}
```
(2) 6-204
```
procedure SetCtrlAttr(Color, Mono : Byte);
  {-Set attributes for control characters}
```
(2) 6-205
```
procedure SetHelpIndex(Index : Word);
  {-Set the help index}
```
(2) 6-204
```
procedure SetCommandProcessor(var CP : CommandProcessor);
  {-Set command processor to use}
```

OpDate Procedures and Functions

(2) 6-227
```
function ValidDate(Day, Month, Year : Integer) : Boolean;
  {-Verify that day, month, year is a valid date}
```
(2) 6-217
```
function DMYtoDate(Day, Month, Year : Integer) : Date;
  {-Convert from day, month, year to a julian date}
```
(2) 6-215
```
procedure DateToDMY(Julian : Date; var Day, Month, Year : Integer);
  {-Convert from a julian date to day, month, year}
```
(2) 6-218
```
function IncDate(Julian : Date;
                 Days, Months, Years : Integer) : Date;
  {-Add (or subtract) the number of days, months, and years to a
    date}
```

```
(2) 6-219    function IncDateTrunc(Julian : Date; Months, Years : Integer) : Date;
               {-Add (or subtract) the specified number of months and years to a
                 date}
(2) 6-212    procedure DateDiff(Date1, Date2 : Date;
                                var Days, Months, Years : Integer);
               {-Return the difference in days,months,years between two valid
                 julian dates}
(2) 6-215    function DayOfWeek(Julian : Date) : DayType;
               {-Return the day of the week for the date}
(2) 6-216    function DayOfWeekDMY(Day, Month, Year : Integer) : DayType;
               {-Return the day of the week for the day, month, year}
(2) 6-213    function DateStringToDate(Picture, S : DateString) : Date;
               {-Convert S, a string of the form indicated by Picture, to a
                 julian date. Picture and S must be of equal lengths}
(2) 6-214    function DateToDateString(Picture : DateString;
                                       Julian : Date) : DateString;
               {-Convert Julian to a string of the form indicated by Picture}
(2) 6-226    function Today : Date;
               {-Returns today's date as a julian}
(2) 6-215    function DateToSortString(Julian : Date) : string;
               {-Convert a date to a sortable string}
(2) 6-222    function SortStringToDate(S : string) : Date;
               {-Convert a sortable string form to a date}
(2) 6-221    function IsLeapYear(Year : Integer) : Boolean;
               {-Return True if Year is a leap year}
(2) 6-216    function DaysInMonth(Month, Year : Integer) : Integer;
               {-Return the number of days in the specified month of a given year}
(2) 6-213    function DateStringToDMY(Picture, S : DateString;
                                      var D, M, Y : Integer) : Boolean;
               {-Extract day, month, and year from S, returning true if string is
                 valid}
(2) 6-217    function DMYtoDateString(Picture : DateString;
                                      Day, Month, Year : Integer) : DateString;
               {-Merge the month, day, and year into the picture}
(2) 6-227    function TodayString(Picture : DateString) : DateString;
               {-Returns today's date as a string of the specified form}
(2) 6-227    function ValidTime(Hours, Minutes, Seconds : Integer) : Boolean;
               {-Return true if Hours:Minutes:Seconds is a valid time}
(2) 6-225    procedure TimeToHMS(T : Time; var Hours, Minutes, Seconds : Byte);
               {-Convert a Time variable to Hours, Minutes, Seconds}
(2) 6-217    function HMStoTime(Hours, Minutes, Seconds : Byte) : Time;
               {-Convert Hours, Minutes, Seconds to a Time variable}
(2) 6-224    function TimeStringToHMS(Picture, St : DateString;
                                      var H, M, S : Integer) : Boolean;
               {-Extract Hours, Minutes, Seconds from St, returning True if
                 string is valid}
(2) 6-224    function TimeStringToTime(Picture, S : DateString) : Time;
               {-Convert S, a string of the form indicated by Picture, to Time}
(2) 6-226    function TimeToTimeString(Picture : DateString;
                                       T : Time) : DateString;
               {-Convert T to a string of the form indicated by Picture}
(2) 6-225    function TimeToAmPmString(Picture : DateString;
                                       T : Time) : DateString;
               {-Convert T to a string of the form indicated by Picture. Times are
                 always displayed in am/pm format.}
(2) 6-212    function CurrentTime : Time;
               {-Returns current time in seconds since midnight}
(2) 6-212    function CurrentTimeString(Picture : DateString) : DateString;
               {-Returns current time as a string of the specified form}
(2) 6-223    procedure TimeDiff(Time1, Time2 : Time;
                                var Hours, Minutes, Seconds : Byte);
               {-Return the difference in hours,minutes,seconds between two times}
(2) 6-219    function IncTime(T : Time; Hours, Minutes, Seconds : Byte) : Time;
               {-Add the specified hours,minutes,seconds to T and return result}
```

```
(2) 6-216    function DecTime(T : Time; Hours, Minutes, Seconds : Byte) : Time;
               {-Subtract the specified hours,minutes,seconds from T and return
                 the result}
(2) 6-226    function TimeToSortString(T : Time) : string;
               {-Convert a time variable to a sortable string}
(2) 6-221    function RoundToNearestHour(T : Time; Truncate : Boolean) : Time;
               {-Round T to the nearest hour or Truncate minutes & seconds from T}
(2) 6-222    function RoundToNearestMinute(T : Time; Truncate : Boolean) : Time;
               {-Round T to the nearest minute, or Truncate seconds from T}
(2) 6-223    function SortStringToTime(S : string) : Time;
               {-Convert a sortable string to a time variable}
(2) 6-214    procedure DateTimeDiff(DT1, DT2 : DateTimeRec;
                                    var Days : Word; var Secs : LongInt);
               {-Return the difference in days and seconds between two points
                 in time}
(2) 6-218    procedure IncDateTime(var DT1, DT2 : DateTimeRec; Days : Integer;
                                   Secs : LongInt);
               {-Increment (or decrement) DT1 by the specified number of days
                 and seconds and put the result in DT2}
(2) 6-220    function InternationalDate(WholeYear,
                                        ZeroPad : Boolean) : DateString;
               {-Return a picture mask for a date string, based on DOS's
                 country info}
(2) 6-220    function InternationalTime(
                 WithSeconds, ZeroPad, ExtraSpace, WithEm : Boolean) : DateString;
               {-Return a picture mask for a time string based on DOS's
                 country info}
```

OpPrnLow BasePrinter

```
(2) 7-19     constructor Init(PrnName : PathStr; TypeOfPrinter : PrnType;
                              PrnFlags : Word);
               {-Initializes a printer object}
(2) 7-15     procedure PrnPutChar(Character : Char); Virtual;
               {-Puts a character to the output device}
(2) 7-14     procedure PrnPutBlock(var Block; BlockSize : Word); Virtual;
               {-Puts a block of characters to the output device}
2) 7-16     function PrnStatus : Word; Virtual;
               {-Return the status of the device}
(2) 7-15     procedure PrnReset; Virtual;
               {-Reset the output device}
(2) 7-13     procedure PrnFlush; Virtual;
               {-Flush the device}
(2) 7-16     function PrnXlatErrorCode(Call : PrnCallType;
                                       ErrorCode : Word) : Word; Virtual;
               {-Translate a raw error code into appropriate user error code}
(2) 7-10     function CharXlat(C : Char) : Char; Virtual;
               {-Translate the Character C according to built in translation
                 table}
(2) 7-10     destructor Done; Virtual;
               {-Flush, close, and deallocate memory used by a printer object}
(2) 7-18     function TypeOfPrn : PrnType;
               {-Returns the type of printer}
(2) 7-11     function NameOfPrn : String;
               {-Returns the printer name}
(2) 7-12     function PrnError  : Word;
               {-Returns the last error and resets pLastError to zero}
(2) 7-15     function PrnRawError : Word;
               {-Returns the raw error code returned by the last operation}
(2) 7-13     function PrnErrorOp : PrnCallType;
               {-Returns the type of the last operation}
(2) 7-14     function PrnLastPut : Word;
               {-Returns the number of characters successfully written by last
                 PrnPutBlock}
```

```
(2) 7-18    procedure XlatCharItem(CharIndex : Char; Character : Char);
              {-Change the character associated with the ascii code of CharIndex
                to Character}
(2) 7-10    procedure ClearXlatCharTable;
              {-Clears the character translation table (the default state)}
(2) 7-12    procedure NewXlatCharTable(var Table : XlatCharTable);
              {-Sets entire XlatChar table}
(2) 7-11    constructor Load(var S : IdStream);
              {-Load a BasePrinter from a stream}
(2) 7-17    procedure Store(var S : IdStream);
              {-Store a BasePrinter in a stream}
```

OpPrnLow DosPrinter

```
(2) 7-19    constructor Init(PrnName : PathStr; TypeOfPrinter : PrnType;
                             PrnFlags : Word);
              {-Initializes a printer object}
```

OpPrnLow BiosPrinter

```
(2) 7-21    constructor Init(LPTNumber : LPTType);
              {-Initializes a BIOS printer object}
(2) 7-22    constructor InitCustom(LPTNumber : LPTType;
                                   PrinterTestNumber, SuccessMask : Byte);
              {-Initializes a BIOS printer object with custom status check}
(2) 7-20    function GetLptPort : LPTType;
              {-Returns the LPT port number for this printer}
(2) 7-22    procedure SetLptPort(LPTNo : LPTType);
              {-Sets the LPT port number for this printer}
(2) 7-23    procedure SetXlatErrorFunc(Xlat : XlatFunc);
              {-Sets the XlatError routine}
(2) 7-21    function GetXlatErrorFunc : Pointer;
              {-Returns the Xlat Error routine}
(2) 7-20    procedure GetTestAndMask(var PrinterTestNumber, Mask : Byte);
              {-Return the printer test number and success mask}
(2) 7-23    procedure SetTestAndMask(PrinterTestNumber, Mask : Byte);
              {-Set the printer test number and success mask}
```

OpPrnLow Procedures and Functions

```
(2) 7-26    procedure DefaultXlatCharTable(var Table : XlatCharTable);
              {-Initialize Table to default state}
(2) 7-24    procedure AssignPrnObject(var F : Text;
                                      var PrinterObj : BasePrinter);
              {-Assign Text variable with a printer object}
(2) 7-26    function GetPrnObject(var F : Text) : BasePrinterPtr;
              {-Return a pointer to the Prn Object associated with a Text file}
(2) 7-25    function CallPrnStatus(var F : Text) : Word;
              {-Call PrnStatus method for Text file associated with Prn Object}
```

OpPrint Printer

```
(2) 7-44    constructor Init(Name : String; DeviceType : PrnType;
                             OptionFlags : Word);
              {-Initialize a Printer}
(2) 7-45    constructor InitCustom(Device : BasePrinterPtr;
                                   LPI, CPI, Point, LPP : Dimension;
                                   OptionFlags : Word);
              {-Initialize a printer with full control over Printer Device,
                PrinterModes}
(2) 7-40    destructor Done; Virtual;
              {-Destroy a printer}
(2) 7-50    procedure PrintStr(S : String); Virtual;
              {-Print a String at the current position}
```

```
(2) 7-50    procedure PrintStrPos(S : String; Row, Col : Dimension;
                                  Relative : RelativeType); Virtual;
              {-Print a string at the specified position}
(2) 7-49    procedure Position(Row, Col : Dimension;
                              Relative : RelativeType); Virtual;
              {-Position the printer to the desired place on the page}
(2) 7-41    procedure FormFeed;
              {-Advance to next page}
(2) 7-39    procedure AdvanceLine(NumLines : Byte);
              {-Advances NumLines number of lines}
(2) 7-39    procedure AdvanceSingleLine; Virtual;
              {-Advances to the top of the next line}
(2) 7-51    procedure ResetPage;
              {-Reset position information}
(2) 7-52    procedure SetLPP(LPP : Dimension);
              {-Sets lines per page}
(2) 7-54    procedure SetYRatio(Ratio : Dimension);
              {-Sets the Y printer ratio}
(2) 7-53    procedure SetXRatio(Ratio : Dimension);
              {-Sets the X printer ratio}
(2) 7-42    function GetLPP : Dimension;
              {-Returns lines per page}
(2) 7-43    function GetYRatio : Dimension;
              {-Returns the Y printer ratio}
(2) 7-43    function GetXRatio : Dimension;
              {-Returns the X printer ratio}
(2) 7-43    function GetRegisteredType : Word;
              {-Returns the hardware type registered to this Printer}
(2) 7-41    function GetBasePrinter : BasePrinterPtr;
              {-Returns the BasePrinterPtr}
(2) 7-41    function GetCurrentFont : Byte;
              {-Returns the current font's Mode ID or zero if no current font}
(2) 7-53    procedure SetPrintErrorFunc(PEF : PrintErrorFunc);
              {-Set the print error function}
(2) 7-53    procedure SetPositionPrim(PosFunc : PrinterPosFunc);
              {-Set printer position primitive}
(2) 7-38    procedure AddPrintMode(Name, OnEscSeq, OffEscSeq : String;
                                   Width, Height : Dimension;
                                   ID : Byte; OptionFlags : Word);
              {-Add a Print Mode to the linked list of modes}
(2) 7-51    procedure SetDefaultFont(Attr : Byte);
              {-Sets the default font for this printer}
(2) 7-42    function GetDefaultFont : Byte;
              {-Returns the Attr of the default font}
(2) 7-40    procedure EnableDefaultFont;
              {-Enables the default font}
(2) 7-56    procedure TurnOnByID(ID : Byte);
              {-Turn a print mode on by ID}
(2) 7-55    procedure TurnOffByID(ID : Byte);
              {-Turn a print mode off by ID}
(2) 7-56    procedure TurnOnByName(Name : String);
              {-Turn a print mode on by Name}
(2) 7-56    procedure TurnOffByName(Name : String);
              {-Turn a print mode off by Name}
(2) 7-49    function PrinterReady : Boolean;
              {-Returns true if printer ready}
(2) 7-48    procedure pOptionsOn(OptionFlags : Word);
              {-Turn on multiple Printer options}
(2) 7-48    procedure pOptionsOff(OptionFlags : Word);
              {-Turn off multiple printer options}
(2) 7-48    function pOptionsAreOn(OptionFlags : Word) : Boolean;
              {-Return True if all of the specified OptionFlags are on}
(2) 7-47    function pGetLastError : Word;
              {-Return the last error code and reset to zero}
```

(2) 7-45 function InLandscape : Boolean;
{-Returns true if in Landscape mode}
(2) 7-55 procedure TurnOffAllActiveModes;
{-Turn off all the currently active print modes}
(2) 7-39 procedure DisposeAllModes;
{-Disposes of all print modes associated with this printer}
(2) 7-47 constructor Load(var S : IdStream);
{-Load a Printer from a stream}
(2) 7-54 procedure Store(var S : IdStream);
{-Store a Printer in a stream}
(2) 7-46 procedure LinkToPrinter(Device : BasePrinterPtr);
{-Links the object with the specified Device}
(2) 7-46 procedure LinkToModes(var Modes : SingleList);
{-Links the object to the specified Print Mode List}

OpDevice Procedures and Functions

(2) 7-63 procedure Deregister(P : PrinterPtr);
{-Deregisters a Printer of any previous registration}
(2) 7-65 procedure LJRegister(P : PrinterPtr);
{-Register this printer as an HP Laserjet compatible}
(2) 7-65 procedure LJFontGRegister(P : PrinterPtr);
{-Registers cartridge font G for LJ}
(2) 7-64 function LJDeci2Row(Decipoints : Dimension;
P : PrinterPtr) : Dimension;
{-Convert decipoint to Row number}
(2) 7-64 function LJDeci2Col(Decipoints : Dimension;
P : PrinterPtr) : Dimension;
{-Convert decipoint to Col number}
(2) 7-64 function LJRow2Deci(Row : Dimension; P : PrinterPtr) : Dimension;
{-Convert a Row number to a Decipoint}
(2) 7-63 function LJCol2Deci(Col : Dimension; P : PrinterPtr) : Dimension;
{-Convert a Row number to a Decipoint}
(2) 7-65 procedure DiabloRegister(P : PrinterPtr);
{-Register this printer as a Diablo compatible}
(2) 7-65 procedure EpsonMXRegister(P : PrinterPtr);
{-Registers P as an Epson MX compatible printer}
(2) 7-65 procedure EpsonLQRegister(P : PrinterPtr);
{-Registers P as an Epson LQ printer}
(2) 7-65 procedure ProPrinterIIRegister(P : PrinterPtr);
{-Registers P as an IBM Proprinter II printer}
(2) 7-65 procedure GenericEpsonRegister(P : PrinterPtr);
{-Registers P as a Generic Epson compatible printer}
(2) 7-65 procedure HPDeskJetRegister(P : PrinterPtr);
{-Registers P as an HP Deskjet compatible}
(2) 7-65 procedure KXP1093Register(P : PrinterPtr);
{-Registers P as a Panasonic KX-P1093 printer}
(2) 7-65 procedure ToshibaP321Register(P : PrinterPtr);
{-Registers P as Toshiba P321SL or P341SL}
(2) 7-65 procedure StarMicronicsRegister(P : PrinterPtr);
{-Registers P as a Star Micronics printer}
(2) 7-65 procedure Nec3510Register(P : PrinterPtr);
{-Registers P as a Nec 3510/3530 printer}
(2) 7-65 procedure Nec8800Register(P : PrinterPtr);
{-Registers P as a Nec 7710/7730/8830/8810 printer}
(2) 7-65 procedure ProwriterRegister(P : PrinterPtr);
{-Registers P as a C. Itoh 8510 Prowriter printer}
(2) 7-65 procedure FacitRegister(P : PrinterPtr);
{-Registers P as a Facit 4512 printer}
(2) 7-65 procedure DataProductsRegister(P : PrinterPtr);
{-Registers P as a DataProducts DP printer}
(2) 7-65 procedure LJDeviceStream(SPtr : IdStreamPtr);
{-Registers the LJ specific procedure and function pointers}

OpForm PrintField

(2) 7-89
```
procedure pfOptionsOn(OptionFlags : LongInt);
  {-Activate multiple options}
```
(2) 7-89
```
procedure pfOptionsOff(OptionFlags : LongInt);
  {-Deactivate multiple options}
```
(2) 7-89
```
function pfOptionsAreOn(OptionFlags : LongInt) : Boolean;
  {-Determine if multiple options are on}
```
(2) 7-88
```
function Convert : Boolean; Virtual;
  {-Convert from native type to printable string}
```
(2) 7-90
```
procedure Print(PrinterP : PrinterPtr); Virtual;
  {-Print the print field}
```
(2) 7-91
```
procedure SetConversion(ConvertFunc : PrintConversionFunc);
  {-Sets the conversion routine}
```
(2) 7-88
```
function GetFieldAttr : Byte;
  {-Return the print attribute for this Field}
```
(2) 7-88
```
procedure GetFieldPos(var Row, Col : Dimension);
  {-Return the row and col for this field}
```
(2) 7-91
```
procedure SetFieldPos(Row, Col : Dimension);
  {-Sets the row and col for this field}
```

OpForm PrintedForm

(2) 7-99
```
constructor Init(FormID : Byte; PrinterP : PrinterPtr);
  {-Initialize a PrintedForm}
```
(2) 7-100
```
constructor InitCustom(FormID : Byte; TopMargin, BottomMargin,
                       LeftMargin, RightMargin : Dimension;
                       PrinterP : PrinterPtr);
  {-Initialize a PrintedForm with custom options}
```
(2) 7-92
```
destructor Done; Virtual;
  {-Dispose of a PrintedForm}
```
(2) 7-102
```
procedure Process; Virtual;
  {-Print the Form}
```
(2) 7-97
```
function fPrinterPtr : PrinterPtr;
  {-Returns the PrinterPtr for this form}
```
(2) 7-94
```
function fGetLastError : Word;
  {-Returns the last error and resets to zero}
```
(2) 7-101
```
procedure NewBlock(BaseRow, BaseCol : Dimension;
                   BlockFunc : UserBlockFunc);
  {-Starts a new block}
```
(2) 7-107
```
procedure SetUserRecord(var UserRec; UserRecSize : Word);
  {-Set the address and size of the user record for current block}
```
(2) 7-99
```
function GetUserRecord : Pointer;
  {-Return a pointer to the user record for current block}
```
(2) 7-99
```
function GetUserRecordSize : Word;
  {-Return the size of the user record for current block}
```
(2) 7-93
```
procedure DupCurrentBlock(NewRow, NewCol : Dimension);
  {-Duplicates the current block at the new block coordinates}
```
(2) 7-93
```
procedure DupCurrentPage;
  {-Duplicates the current page}
```
(2) 7-104
```
procedure PromptOptionsOn(OptionFlags : LongInt);
  {-Turn options for the prompts on}
```
(2) 7-95
```
procedure FieldOptionsOn(OptionFlags : LongInt);
  {-Turn options for the fields on}
```
(2) 7-103
```
procedure PromptOptionsOff(OptionFlags : LongInt);
  {-Turn options for the prompt off}
```
(2) 7-95
```
procedure FieldOptionsOff(OptionFlags : LongInt);
  {-Turn options for the fields off}
```
(2) 7-103
```
function PromptOptionsAreOn(OptionFlags : LongInt) : Boolean;
  {-Determine whether options for the prompts are on}
```
(2) 7-95
```
function FieldOptionsAreOn(OptionFlags : LongInt) : Boolean;
  {-Determine whether options for the fields are on}
```

```
(2) 7-72      procedure AddTextField(pFieldID : Word; Prompt : String;
                                     pRow, pCol : Dimension;
                                     pAttr, pWidth : Byte);
                {-Adds a text field (no user variable)}
(2) 7-73      procedure AddStringField(pFieldID : Word; Prompt : String;
                                       pRow, pCol : Dimension;
                                       pAttr, pWidth : Byte;
                                       fFieldID : Word; Picture : String;
                                       fRow, fCol : Dimension;
                                       fAttr, fWidth : Byte;
                                       var PrintSt : String);
                {-Adds a string field}
(2) 7-73      procedure AddCharField(pFieldID : Word; Prompt : String;
                                     pRow, pCol : Dimension;
                                     pAttr, pWidth : Byte;
                                     fFieldID : Word; Picture : String;
                                     fRow, fCol : Dimension;
                                     fAttr, fWidth : Byte;
                                     var PrintChar : Char);
                {-Adds prompt and character field}
(2) 7-72      procedure AddBooleanField(pFieldID : Word; Prompt : String;
                                        pRow, pCol : Dimension;
                                        pAttr, pWidth : Byte;
                                        fFieldID : Word; Picture : String;
                                        fRow, fCol : Dimension;
                                        fAttr, fWidth : Byte;
                                        var PrintBool : Boolean);
                {-Adds a Boolean field}
(2) 7-73      procedure AddYesNoField(pFieldID : Word; Prompt : String;
                                      pRow, pCol : Dimension;
                                      pAttr, pWidth : Byte;
                                      fFieldID : Word; Picture : String;
                                      fRow, fCol : Dimension;
                                      fAttr, fWidth : Byte;
                                      var PrintYesNo : Boolean);
                {-Adds a Yes/No field}
(2) 7-73      procedure AddLongField(pFieldID : Word; Prompt : String;
                                     pRow, pCol : Dimension;
                                     pAttr, pWidth : Byte;
                                     fFieldID : Word; Picture : String;
                                     fRow, fCol : Dimension;
                                     fAttr, fWidth : Byte;
                                     var PrintLong : LongInt);
                {-Adds a LongInt field}
(2) 7-73      procedure AddWordField(pFieldID : Word; Prompt : String;
                                     pRow, pCol : Dimension;
                                     pAttr, pWidth : Byte;
                                     fFieldID : Word; Picture : String;
                                     fRow, fCol : Dimension;
                                     fAttr, fWidth : Byte;
                                     var PrintWord : Word);
                {-Adds a Word field}
(2) 7-73      procedure AddIntField(pFieldID : Word; Prompt : String;
                                    pRow, pCol : Dimension;
                                    pAttr, pWidth : Byte;
                                    fFieldID : Word; Picture : String;
                                    fRow, fCol : Dimension;
                                    fAttr, fWidth : Byte;
                                    var PrintInt : Integer);
                {-Adds an Integer field}
(2) 7-73      procedure AddByteField(pFieldID : Word; Prompt : String;
                                     pRow, pCol : Dimension;
                                     pAttr, pWidth : Byte;
                                     fFieldID : Word; Picture : String;
                                     fRow, fCol : Dimension;
                                     fAttr, fWidth : Byte;
                                     var PrintByte : Byte);
                {-Adds a Byte field}
```

```
(2) 7-73    procedure AddShortField(pFieldID : Word; Prompt : String;
                                    pRow, pCol : Dimension;
                                    pAttr, pWidth : Byte;
                                    fFieldID : Word; Picture : String;
                                    fRow, fCol : Dimension;
                                    fAttr, fWidth : Byte;
                                    var PrintShort : ShortInt);
              {-Adds a ShortInt field}
(2) 7-73    procedure AddRealField(pFieldID : Word; Prompt : String;
                                   pRow, pCol : Dimension;
                                   pAttr, pWidth : Byte;
                                   fFieldID : Word; Picture : String;
                                   fRow, fCol : Dimension;
                                   fAttr : Byte; fDPlaces : Byte;
                                   fWidth : Byte; var PrintReal : Real);
              {-Adds a Real field}
(2) 7-74    procedure AddDateField(pFieldID : Word; Prompt : String;
                                   pRow, pCol : Dimension;
                                   pAttr, pWidth : Byte;
                                   fFieldID : Word; Picture : String;
                                   fRow, fCol : Dimension;
                                   fAttr, fWidth : Byte;
                                   var PrintDate : Date);
              {-Adds a Date field}
(2) 7-74    procedure AddTimeField(pFieldID : Word; Prompt : String;
                                   pRow, pCol : Dimension;
                                   pAttr, pWidth : Byte;
                                   fFieldID : Word; Picture : String;
                                   fRow, fCol : Dimension;
                                   fAttr, fWidth : Byte;
                                   var PrintTime : Time);
              {-Adds a Time field}
(2) 7-75    procedure AddDirective(FieldID : Word; Row, Col : Dimension;
                                   On : Boolean; Attr : Byte);
              {-Adds a print directive}
(2) 7-75    procedure AddLine(FieldID : Word; Row, Col : Dimension;
                              Len, LineThickness : Dimension;
                              Horizontal : Boolean; LineChar : Char;
                              Attr : Byte);
              {-Adds a line field}
(2) 7-75    procedure AddBox(FieldID : Word; Row, Col : Dimension;
                             Width, Height : Dimension;
                             HLineThickness, VLineThickness : Dimension;
                             BoxChars : BoxCharSet; Attr : Byte);
              {-Adds a box field}
(2) 7-75    procedure AddShaded(FieldID : Word; Row, Col : Dimension;
                                Width, Height : Dimension; Intensity : Byte);
              {-Adds a shaded box field}
(2) 7-102   procedure NewPage;
              {-Start a new page}
(2) 7-106   procedure SetHeader(HeaderStr : String; Row, Col : Dimension;
                                HAttr : Byte);
              {-Set up a header for the current page}
(2) 7-106   procedure SetHeaderFunc(Picture : String;
                                    HeadFunc : PrintConversionFunc);
              {-Set the HeadFunc function pointer for the current page}
(2) 7-98    function GetHeader : String;
              {-Returns the header string for the current page}
(2) 7-105   procedure SetFooter(FooterStr : String; Row, Col : Dimension;
                                FAttr : Byte);
              {-Set up a Footer for the current page}
(2) 7-105   procedure SetFooterFunc(Picture : String;
                                    FootFunc : PrintConversionFunc);
              {-Set the HeadFunc function pointer for the current page}
(2) 7-97    function GetFooter : String;
              {-Returns the Footer string for the current page}
(2) 7-104   procedure SetErrorHandler(ErrorFunc : PrintErrorFunc);
              {-Sets the user error handler}
```

(2) 7-96
```
function FindField(FieldID : Word; var Page : PrintPagePtr;
                   var Block : PrintBlockPtr) : PrintFieldPtr;
  {-Finds the first field with the specified FieldID}
```
(2) 7-96
```
function FindNextField(Field  : PrintFieldPtr;
                       var Page : PrintPagePtr;
                       var Block : PrintBlockPtr) : PrintFieldPtr;
  {-Finds the next field with the same FieldID as Field}
```
(2) 7-92
```
procedure ChangeConversion(FieldID : Word;
                           ConvertFunc : PrintConversionFunc);
  {-Changes field's conversion routine}
```
(2) 7-107
```
procedure SetUserData(P : Pointer);
  {-Sets the user data field}
```
(2) 7-98
```
function GetUserData : Pointer;
  {-Returns the user data field}
```
(2) 7-106
```
procedure SetPrePageFunc(ppFunc : UserPageFunc);
  {-Sets the function that gets called before the current page is
    printed}
```
(2) 7-107
```
procedure SetPrePostFormFunc(ppFunc : UserFormFunc);
  {-Sets the function that gets called before and after printing
    the form}
```
(2) 7-100
```
constructor Load(var S : IdStream);
  {-Load a PrintedForm from a stream}
```
(2) 7-108
```
procedure Store(var S : IdStream);
  {-Store a PrintedForm in a stream}
```

OpForm Report

(2) 7-110
```
constructor Init;
  {-Create a report}
```
(2) 7-109
```
destructor Done; Virtual;
  {-Destroy a report}
```
(2) 7-112
```
function rGetLastError : Word;
  {-Get last error code and clear error code}
```
(2) 7-111
```
procedure Process; Virtual;
  {-Print the print field}
```
(2) 7-109
```
procedure AddForm(fPtr : PrintedFormPtr);
  {-Add a PrintedForm to Report by pointer}
```
(2) 7-110
```
function FindForm(FormID : Byte) : PrintedFormPtr;
  {-Return a pointer to the PrintedForm with the specified ID}
```
(2) 7-111
```
constructor Load(var S : IdStream);
  {-Load a Report from a stream}
```
(2) 7-112
```
procedure Store(var S : IdStream);
  {-Store a Report in a stream}
```

OpRoot Root

(3) 8-99
```
constructor Init;
  {-Initialize the object}
```
(3) 8-99
```
destructor Done; virtual;
  {-Destroy the object}
```

OpRoot IdStream

(3) 8-108
```
function ReadString : String;
  {-Read next variable length string from stream}
```
(3) 8-107
```
procedure ReadRange(var BufStart, BufBeyond);
  {-Read Ofs(BufBeyond)-Ofs(BufStart) bytes into BufStart}
```
(3) 8-118
```
procedure WriteString(S : String);
  {-Write a variable length string to stream}
```
(3) 8-118
```
procedure WriteRange(var BufStart, BufBeyond);
  {-Write Ofs(BufBeyond)-Ofs(BufStart) bytes to stream}
```
(3) 8-102
```
function GetSize : Longint;
  {-Determine size of stream}
```

(3) 8-116
```
procedure Seek(Pos : Longint);
  {-Set the current stream position to Pos,
    relative to the beginning of stream}
```
(3) 8-101
```
function GetPos : Longint; virtual;
  {-Return the current stream position}
```
(3) 8-103
```
function GetStatus : Word;
  {-Return status of stream}
```
(3) 8-103
```
function PeekStatus : Word;
  {-Return status and leave internal variable unchanged}
```
(3) 8-115
```
procedure RegisterType(Code, Version : Word;
                       TypePtr, StorePtr, LoadPtr : Pointer);
  {-Registers one object type of specified version for IdStream I/O.
    May set Status to non-zero}
```
(3) 8-111
```
procedure RegisterOldVersion(Code, Version : Word;
                             TypePtr, LoadPtr : Pointer);
  {-Registers an alternate loader for an old version of an object
    type}
```
(3) 8-116
```
function TypeRegistered(TypePtr : Pointer) : Boolean;
  {-Return True if specified type is registered for store and load}
```
(3) 8-117
```
function TypeVerRegistered(TypePtr : Pointer;
                           Version : Word) : Boolean;
  {-Return True if specified type and version is registered for load}
```
(3) 8-100
```
function CodeVerRegistered(Code, Version : Word) : Boolean;
  {-Return True if specified code and version number are registered}
```
(3) 8-110
```
procedure RegisterHier(RProc : StreamRegisterProc);
  {-Call a supplied routine to register a hierarchy of object types}
```
(3) 8-105
```
procedure PutPtr(RPtr : RootPtr);
  {-Calls polymorphic Store method to write object at RPtr^}
```
(3) 8-102
```
function GetPtr : RootPtr;
  {-Calls polymorphic Load constructor to allocate and read object}
```
(3) 8-104
```
procedure Put(var R : Root);
  {-Calls polymorphic Store method to write object R}
```
(3) 8-101
```
procedure Get(var R : Root);
  {-Calls polymorphic Load constructor to read static object R}
```
(3) 8-112
```
procedure RegisterPointer(Code : Word; Ptr : Pointer);
  {-Registers one pointer for IdStream I/O}
```
(3) 8-104
```
function PointerRegistered(Ptr : Pointer) : Boolean;
  {-Return True if specified pointer is registered with stream}
```
(3) 8-117
```
procedure WritePointer(Ptr : Pointer);
  {-Writes identifier for pointer to stream}
```
(3) 8-119
```
procedure WriteUserPointer(Ptr : Pointer; DefaultCode : Word);
  {-Writes Ptr code if registered, else DefaultCode}
```
(3) 8-106
```
function ReadPointer : Pointer;
  {-Reads pointer from stream}
```
(3) 8-109
```
function ReadUserPointer(DefaultPtr : Pointer) : Pointer;
  {-Reads pointer and assigns default instead of nil}
```
(3) 8-100
```
procedure Error(Code : Word); virtual;
  {-Report an error}
```

OpRoot DosIdStream

(3) 8-120
```
constructor Init(FileName : String; Mode : Word);
  {-Open DOS file in specified mode and initialize stream}
```
(3) 8-120
```
destructor Done; virtual;
  {-Close file and dispose stream}
```
(3) 8-121
```
procedure Read(var Buf; Count : Word); virtual;
  {-Read bytes from DOS file}
```
(3) 8-121
```
procedure Write(var Buf; Count : Word); virtual;
  {-Write bytes to DOS file}
```
(3) 8-121
```
procedure Truncate; virtual;
  {-Truncate file at current position}
```

OpRoot BufIdStream

(3) 8-124
```
constructor Init(FileName : String; Mode, Size : Word);
  {-Open DOS file in mode, and allocate buffer of Size bytes}
```
(3) 8-123
```
destructor Done; virtual;
  {-Flush and close file, deallocate buffer, dispose stream}
```
(3) 8-123
```
procedure Flush; virtual;
  {-Flush buffer}
```
(3) 8-124
```
procedure Read(var Buf; Count : Word); virtual;
  {-Read, using buffer}
```
(3) 8-124
```
procedure Write(var Buf; Count : Word); virtual;
  {-Write, using buffer}
```

OpRoot MemIdStream

(3) 8-125
```
constructor Init(BasePtr : Pointer; Size : Word);
  {-Initialize stream to point to a location in memory}
```
(3) 8-126
```
procedure Read(var Buf; Count : Word); virtual;
  {-Read bytes from memory stream}
```
(3) 8-126
```
procedure Write(var Buf; Count : Word); virtual;
  {-Write bytes to memory stream}
```
(3) 8-126
```
procedure Truncate; virtual;
  {-Truncate at current position. Does nothing but reduce Size.}
```

OpRoot DirEntry

(3) 8-135
```
constructor Init;
  {-Initialize an empty directory entry}
```
(3) 8-135
```
procedure Rename(Name : DirEntryName);
  {-Change the name of the directory entry}
```
(3) 8-136
```
procedure Update(Start, Len : LongInt; Code, Ver : Word); virtual;
  {-Update the start and length data}
```
(3) 8-135
```
procedure SetUserPointer(P : Pointer);
  {-Set the user pointer to P}
```
(3) 8-132
```
procedure deOptionsOn(Options : Word);
  {-Set the specified flag(s)}
```
(3) 8-132
```
procedure deOptionsOff(Options : Word);
  {-Clear the specified flag(s)}
```
(3) 8-132
```
function deOptionsAreOn(Options : Word) : Boolean;
  {-Returns True if the specified flag(s) are on}
```
(3) 8-133
```
function GetEntryName : DirEntryName;
  {-Return the name of the entry}
```
(3) 8-134
```
function GetEntryStart : LongInt;
  {-Return the starting point for the entry}
```
(3) 8-133
```
function GetEntryLength : LongInt;
  {-Return the length of the entry}
```
(3) 8-134
```
function GetUserPointer : Pointer;
  {-Return the user pointer}
```

OpRoot Library

(3) 8-143
```
constructor Init(FileName : PathStr; Mode, BufferSize : Word;
                 Sig : LibrarySig);
  {-Open an existing library}
```
(3) 8-137
```
constructor Create(FileName : PathStr; BufferSize : Word;
                   Sig : LibrarySig; Entries : Word);
  {-Create a new library}
```
(3) 8-139
```
destructor Done; virtual;
  {-Flush and close library file, and dispose of directory and
    stream}
```
(3) 8-146
```
procedure PutEntry(Name : DirEntryName; var R : Root);
  {-Calls polymorphic Store method to write object R}
```
(3) 8-142
```
procedure GetEntry(Name : DirEntryName; var R : Root);
  {-Calls polymorphic Load constructor to read static object R}
```

(3) 8-142 function GetEntryPtr(Name : DirEntryName) : RootPtr;
{-Calls polymorphic Load constructor to allocate and read object}

(3) 8-138 procedure DeleteEntry(Name : DirEntryName); virtual;
{-Mark the entry of the specified name as deleted}

(3) 8-147 procedure UndeleteEntry(Name : DirEntryName); virtual;
{-Remove deletion mark from the specified entry}

(3) 8-144 procedure KillEntry(Name : DirEntryName);
{-Permanently delete the entry of the specified name}

(3) 8-147 procedure RenameEntry(OldName, NewName : DirEntryName); virtual;
{-Rename the specified entry}

(3) 8-141 procedure GetCodeAndVersion(Name : DirEntryName;
var Code, Ver : Word);
{-Return the code and version for the specified entry}

(3) 8-145 procedure Pack;
{-Pack the library}

(3) 8-146 function PackedFileSize : LongInt;
{-Returns size of library after packing}

(3) 8-139 function FileNeedsPacking : Boolean;
{-Returns True if library needs packing}

(3) 8-141 function FindEntryByIndex(N : Word) : DirEntryPtr;
{-Return a pointer to the N'th entry in the directory}

(3) 8-140 function FindDirectoryEntry(Name : DirEntryName) : DirEntryPtr;
{-Return a pointer to a directory entry; nil if not found or deleted}

(3) 8-140 function FindAnyDirectoryEntry(Name : DirEntryName) : DirEntryPtr;
{-Return a pointer to a directory entry; nil if not found}

(3) 8-144 function MaxEntries : Word;
{-Return the maximum number of entries in the library's directory}

(3) 8-138 function CurrentEntries : Word;
{-Return the number of used entries in the library's directory, not including deleted entries}

(3) 8-138 function DeletedEntries : Word;
{-Return the number of deleted entries in the library's directory}

(3) 8-137 function AvailableEntries : Word;
{-Return number of entries available in library's directory}

OpRoot MemLibrary

(3) 8-149 constructor Init(BasePtr : Pointer; Size : Word);
{-Open an existing library in memory}

(3) 8-148 procedure GetEntry(Name : DirEntryName; var R : Root);
{-Calls polymorphic Load constructor to read static object R}

(3) 8-148 function GetEntryPtr(Name : DirEntryName) : RootPtr;
{-Calls polymorphic Load constructor to allocate and read object}

OpRoot PointerStack

(3) 8-10 constructor Init(Max : Word);
{-Allocate a stack of pointers that can manage up to Max items}

(3) 8-9 destructor Done; virtual;
{-Deallocate the stack}

(3) 8-16 function Size : Word;
{-Return the size of the stack}

(3) 8-15 procedure Push(P : RootPtr);
{-Push P onto the stack}

(3) 8-14 function Pop : RootPtr;
{-Pop the topmost RootPtr off the stack}

(3) 8-16 function SP : Word;
{-Return the current top of stack index}

(3) 8-13 function Peek(Index : Word) : RootPtr;
{-Return stack element at position Index}

(3) 8-13 function PeekTop : RootPtr;
{-Return pointer currently at the top of stack}

(3) 8-14 procedure Poke(P : RootPtr; Index : Word);
{-Directly set a stack element}

(3) 8-9 procedure Clear;
{-Remove all elements from the stack}
(3) 8-10 function GetStatus : Word;
{-Return latest status code and reset internal status to zero}
(3) 8-16 function Resize(Max : Word) : Boolean;
{-Expand or shrink stack to hold Max pointers}
(3) 8-11 constructor Load(var S : IdStream);
{-Load a stack from a stream}
(3) 8-17 procedure Store(var S : IdStream);
{-Store a stack in a stream}

OpRoot StaticQueue

(3) 8-23 constructor Init(NumElements, ElementSize : Word;
RejectIfFull : Boolean);
{-Allocate space for static queue}
(3) 8-21 destructor Done; virtual;
{-Deallocate space for the queue}
(3) 8-27 procedure PushTail(var Element);
{-Add an element to the tail of the queue}
(3) 8-26 procedure PopTail(var Element);
{-Remove an element from the tail of the queue}
(3) 8-25 procedure PeekTail(var Element);
{-Look at the element at the tail of the queue}
(3) 8-26 procedure PopHead(var Element);
{-Remove an element from the front of the queue}
(3) 8-25 procedure PeekHead(var Element);
{-Look at the element at the front of the queue}
(3) 8-21 procedure Clear;
{-Flush all elements from the queue}
(3) 8-21 function Elements : Word;
{-Return the number of elements currently in the queue}
(3) 8-22 function GetStatus : Word;
{-Return latest status code and reset internal status to zero}
(3) 8-22 procedure Error(Code : Word); virtual;
{-Report error}
(3) 8-24 constructor Load(var S : IdStream);
{-Load a static queue from a stream}
(3) 8-28 procedure Store(var S : IdStream);
{-Store a static queue to a stream}

OpRoot SingleListNode

(3) 8-33 constructor Init;
{-Initialize a singly linked list node}

OpRoot SingleList

(3) 8-35 constructor Init;
{-Initialize an empty list of items}
(3) 8-34 destructor Done; virtual;
{-Destroy list}
(3) 8-39 function Size : Word;
{-Return the size of the current list}
(3) 8-34 procedure Append(P : SingleNodePtr);
{-Add element to end of list}
(3) 8-35 procedure Insert(P : SingleNodePtr);
{-Insert element at start of list}
(3) 8-38 procedure Place(P, L : SingleNodePtr);
{-Place element P into list _after_ existing element L}
(3) 8-39 procedure PlaceBefore(P, L : SingleNodePtr);
{-Place element P into list _before_ existing element L}
(3) 8-34 procedure Delete(P : SingleNodePtr);
{-Delete existing element in list, dispose of its contents}

```
(3) 8-39    procedure Remove(P : SingleNodePtr);
              {-Remove existing element from list without disposing of it}
(3) 8-35    function Head : SingleNodePtr;
              {-Return pointer to head of list}
(3) 8-42    function Tail : SingleNodePtr;
              {-Return pointer to tail of list}
(3) 8-38    function Next(P : SingleNodePtr) : SingleNodePtr;
              {-Returns a pointer to the next node}
(3) 8-36    constructor Load(var S : IdStream);
              {-Load a list from a stream}
(3) 8-40    procedure Store(var S : IdStream);
              {-Store a list in a stream}
```

OpRoot DoubleListNode

```
(3) 8-43    constructor Init;
              {-Initialize a doubly linked list node}
```

OpRoot DoubleList

```
(3) 8-45    constructor Init;
              {-Initialize an empty list of items}
(3) 8-44    destructor Done; virtual;
              {-Destroy list}
(3) 8-48    function Size : Word;
              {-Return the size of the current list}
(3) 8-44    procedure Append(P : DoubleNodePtr);
              {-Add element to end of list}
(3) 8-45    procedure Insert(P : DoubleNodePtr);
              {-Insert element at start of list}
(3) 8-46    procedure Place(P, L : DoubleNodePtr);
              {-Place element P into list _after_ existing element L}
(3) 8-47    procedure PlaceBefore(P, L : DoubleNodePtr);
              {-Place element P into list _before_ existing element L}
(3) 8-44    procedure Delete(P : DoubleNodePtr);
              {-Delete existing element in list, disposing of its contents}
(3) 8-47    procedure Remove(P : DoubleNodePtr);
              {-Remove existing element from list without disposing of it}
(3) 8-45    function Head : DoubleNodePtr;
              {-Return pointer to head of list}
(3) 8-48    function Tail : DoubleNodePtr;
              {-Return pointer to tail of list}
(3) 8-46    function Next(P : DoubleNodePtr) : DoubleNodePtr; virtual;
              {-Returns a pointer to the next node}
(3) 8-47    function Prev(P : DoubleNodePtr) : DoubleNodePtr; virtual;
              {-Returns a pointer to the previous node}
(3) 8-46    constructor Load(var S : IdStream);
              {-Load a list from a stream}
(3) 8-48    procedure Store(var S : IdStream);
              {-Store a list in a stream}
```

OpRoot CircularList

```
(3) 8-49    function Next(P : DoubleNodePtr) : DoubleNodePtr; virtual;
              {-Returns a pointer to the next node (wraps at end)}
(3) 8-49    function Prev(P : DoubleNodePtr) : DoubleNodePtr; virtual;
              {-Returns a pointer to the previous node (wraps at begin)}
(3) 8-49    procedure Store(var S : IdStream);
              {-Store a list in a stream}
```

OpRoot BitSet

```
(3) 8-55    constructor Init(Max : LongInt);
              {-Allocate space for elements}
```

OPROOT

(3) 8-54 destructor Done; virtual;
{-Deallocate data area}
(3) 8-53 procedure ClearAll; virtual;
{-Clear all bits}
(3) 8-60 procedure SetAll; virtual;
{-Set all bits}
(3) 8-58 function MaxBits : LongInt;
{-Return capacity of bitset}
(3) 8-61 procedure SetBit(Element : LongInt); virtual;
{-Set bit}
(3) 8-53 procedure ClearBit(Element : LongInt); virtual;
{-Clear bit}
(3) 8-63 procedure ToggleBit(Element : LongInt);
{-Toggle bit}
(3) 8-54 procedure ControlBit(Element : LongInt; State : Boolean);
{-Set or clear bit depending on State}
(3) 8-63 function TestSetBit(Element : LongInt) : Boolean;
{-Return current state of bit and set it}
(3) 8-63 function TestClearBit(Element : LongInt) : Boolean;
{-Return current state of bit and clear it}
(3) 8-52 function BitIsSet(Element : LongInt) : Boolean; virtual;
{-Return True if bit is set}
(3) 8-59 function NextSet(Element : LongInt) : LongInt; virtual;
{-Return index of next set element, NoMoreBits if none}
(3) 8-58 function NextClear(Element : LongInt) : LongInt; virtual;
{-Return index of next clear element, NoMoreBits if none}
(3) 8-60 function PrevSet(Element : LongInt) : LongInt; virtual;
{-Return index of previous set element, NoMoreBits if none}
(3) 8-59 function PrevClear(Element : LongInt) : LongInt; virtual;
{-Return index of previous clear element, NoMoreBits if none}
(3) 8-52 function BitsSet : LongInt; virtual;
{-Return number of bits set}
(3) 8-55 function FirstSet : LongInt;
{-Return index of first set bit, NoMoreBits if none}
(3) 8-56 function LastSet : LongInt;
{-Return index of last set bit, NoMoreBits if none}
(3) 8-54 function FirstClear : LongInt;
{-Return index of first clear bit, NoMoreBits if none}
(3) 8-56 function LastClear : LongInt;
{-Return index of last clear bit, NoMoreBits if none}
(3) 8-57 constructor Load(var S : IdStream);
{-Load a bitset from a stream}
(3) 8-62 procedure Store(var S : IdStream);
{-Store a bitset to a stream}

OpRoot LargeBitSet

(3) 8-65 constructor Init(Max : LongInt);
{-Allocate space for elements}
(3) 8-66 function Resize(Max : LongInt) : Boolean;
{-Expand or shrink large bitset to hold Max bits}
(3) 8-64 function GetStatus : Word;
{-Return latest status code and reset internal status to zero}
(3) 8-64 procedure Error(Code : Word); virtual;
{-Report an error}
(3) 8-65 constructor Load(var S : IdStream);
{-Load a bitset from a stream}
(3) 8-66 procedure Store(var S : IdStream);
{-Store a bitset to a stream}

OpRoot StringDict

(3) 8-75 constructor Init;
{-Allocate hash pool for a string dictionary}

(3) 8-73 destructor Done; virtual;
{-Deallocate pool}
(3) 8-71 procedure Add(S : String; Value : LongInt);
{-Add string and associated value to dictionary}
(3) 8-77 function Member(S : String; var Value : LongInt) : Boolean;
{-Return true and associated value if S is element of dictionary}
(3) 8-79 procedure Update(S : String; Value : LongInt);
{-Change the data value associated with a string}
(3) 8-77 procedure Remove(S : String);
{-Remove string from dictionary}
(3) 8-73 procedure Clear;
{-Remove all strings from dictionary}
(3) 8-74 function GetStatus : Word;
{-Return status code and reset internal result to zero}
(3) 8-76 constructor Load(var S : IdStream);
{-Load a string dictionary from a stream}
(3) 8-78 procedure Store(var S : IdStream);
{-Store a string dictionary to a stream}
(3) 8-75 function Hash(var S : String) : Word; virtual;
{-Return a hash code for the specified string}
(3) 8-73 function Equal(var S, T : String) : Boolean; virtual;
{-Return True if the two strings are considered equal}
(3) 8-74 procedure Error(Code : Word); virtual;
{-Report an error}

OpRoot StringSet

(3) 8-80 procedure Add(S : String);
{-Add string to set}
(3) 8-81 function Member(S : String) : Boolean;
{-Return true if S is element of set}
(3) 8-80 constructor Load(var S : IdStream);
{-Load a list from a stream}
(3) 8-81 procedure Store(var S : IdStream);
{-Store a list in a stream}

OpRoot StringArray

(3) 8-84 constructor Init(StrMax, Amount : Word);
{-Allocate space for StrMax strings in Amount space}
(3) 8-86 constructor ReadText(FileName : PathStr);
{-Initialize a packed array from a text file}
(3) 8-82 destructor Done; virtual;
{-Deallocate packed array}
(3) 8-89 procedure WriteText(FileName : PathStr);
{-Write a packed array to a text file}
(3) 8-85 constructor Load(var S : IdStream);
{-Load a binary packed array from a stream}
(3) 8-88 procedure Store(var S : IdStream);
{-Write a packed array to a stream}
(3) 8-83 function GetString(Which : Word) : String; virtual;
{-Return string from packed array}
(3) 8-83 function GetStringPtr(Which : Word) : StringPtr;
{-Return pointer to string from packed array}
(3) 8-87 function Shrink : Boolean;
{-Remove excess packed array space}
(3) 8-82 function AddString(St : String) : Word;
{-Add a new string, returning its index, or 0 if error}
(3) 8-85 function NumStrings : Word;
{-Return number of strings in the packed array}
(3) 8-87 function Size : Word;
{-Return the number of bytes used by strings}

OPLARRAY

OpLarray AbstractArray

(3) 8-165
```
constructor Init(Rows, Cols : Word; ElementSize : Word;
                 ArrayOptions : Byte);
  {-Make a new array}
```
(3) 8-168
```
constructor LoadA(ArrayOptions : Byte);
  {-Load an array from disk}
```
(3) 8-162
```
destructor Done; Virtual;
  {-Dispose of an array}
```
(3) 8-163
```
function ErrorA : Word;
  {-Return last error code (0 indicates no error)}
```
(3) 8-173
```
procedure SetA(Row, Col : Word; var Value); Virtual;
  {-Set an array element}
```
(3) 8-171
```
procedure RetA(Row, Col : Word; var Value); Virtual;
  {-Return an array element}
```
(3) 8-172
```
procedure SafeSetA(Row, Col : Word; var Value); Virtual;
  {-Set an array element and safely deal with out of range
    coordinates}
```
(3) 8-172
```
procedure SafeRetA(Row, Col : Word; var Value); Virtual;
  {-Return an array element and safely deal with out of range
    coordinates}
```
(3) 8-161
```
procedure ClearA(var Value; Initialize : InitType); Virtual;
  {-Clear the array}
```
(3) 8-175
```
procedure StoreA(FileName : String); Virtual;
  {-Store large array in a file}
```
(3) 8-164
```
procedure FlushA; Virtual;
  {-Flush changes to disk}
```
(3) 8-170
```
function RangeError(Row, Col : Word) : Boolean; Virtual;
  {-Returns true if passed Row or Col are out of range}
```
(3) 8-175
```
function TypeOfArray : ArrayType;
  {-Returns the type of array}
```
(3) 8-174
```
procedure SetErrorProc(EP : ErrorProc);
  {-Setup a user errorproc}
```
(3) 8-164
```
procedure GetErrorProc(var EP : ErrorProc);
  {-Get current errorproc}
```
(3) 8-161
```
procedure ArrayDimensions(var Rows,Cols : Word);
  {-Return array dimensions}
```
(3) 8-169
```
procedure lOptionsOn(OptionCodes : Word);
  {-Set array options on}
```
(3) 8-169
```
procedure lOptionsOff(OptionCodes : Word);
  {-Set array options off}
```
(3) 8-169
```
function lOptionsAreOn(OptionCodes : Word) : Boolean;
  {-Check options}
```

OpLarray RamArray

(3) 8-165
```
constructor Init(Rows, Cols : Word; ElementSize : Word;
                 ArrayOptions : Byte);
  {-Make a new array}
```
(3) 8-168
```
constructor LoadA(FileName : String; ArrayOptions : Byte);
  {-Load an array from disk}
```

OpLarray RamRCArray

(3) 8-165
```
constructor Init(Rows, Cols : Word; ElementSize : Word;
                 ArrayOptions : Byte);
  {-Make a new array}
```
(3) 8-168
```
constructor LoadA(FileName : String; ArrayOptions : Byte);
  {-Load an array from disk}
```

OpLarray EmsArray

(3) 8-165
```
constructor Init(Rows, Cols : Word; ElementSize : Word;
                 ArrayOptions : Byte);
  {-Make a new array}
```

(3) 8-168 `constructor LoadA(FileName : String; ArrayOptions : Byte);`
`{-Load an array from disk}`

OpLarray EmsRCArray

(3) 8-165 `constructor Init(Rows, Cols : Word; ElementSize : Word;`
`ArrayOptions : Byte);`
`{-Make a new array}`

(3) 8-168 `constructor LoadA(FileName : String; ArrayOptions : Byte);`
`{-Load an array from disk}`

OpLarray VirtualArray

(3) 8-165 `constructor Init(Rows, Cols : Word; ElementSize : Word;`
`FileName : String; HeapToUse : LongInt;`
`ArrayOptions : Byte);`
`{-Make a new array}`

(3) 8-168 `constructor LoadA(FileName : String; ArrayOptions : Byte);`
`{-Load an array from disk}`

OpLarray PrimitiveArray

(3) 8-165 `constructor Init(RowsInPage : Word; ColsInPage : Word;`
`PagesDown : Word; PagesAcross : Word;`
`ElementSize : Word; FileName : string;`
`HeapToUse : LongInt; ArrayOptions : Byte);`
`{-Make a new array}`

OpLarray VirtualRCArray

(3) 8-165 `constructor Init(Rows, Cols : Word; ElementSize : Word;`
`FileName : String; HeapToUse : LongInt;`
`ArrayOptions : Byte);`
`{-Make a new array}`

(3) 8-168 `constructor LoadA(FileName : String; ArrayOptions : Byte);`
`{-Load an array from disk}`

OpLarray OpArray

(3) 8-165 `constructor Init(Rows, Cols : Word; ElementSize : Word;`
`FileName : String; HeapToUse : LongInt;`
`ArrayOptions : Byte; var Priority : AutoPriority);`
`{-Make a new array}`

(3) 8-168 `constructor LoadA(FileName : String; ArrayOptions : Byte;`
`var Priority : AutoPriority);`
`{-Load an array}`

OpLarray Procedures and Functions

(3) 8-176 `procedure DisplayLErrorProc(UnitCode : Byte; var ErrorCode : Word;`
`ErrorMsg : String);`
`{-An error procedure that displays a standard message for errors generated by this unit. Does not halt on errors}`

OpString Procedures and Functions

(3) 9-16 `function HexB(B : Byte) : string;`
`{-Return hex string for byte}`

(3) 9-17 `function HexW(W : Word) : string;`
`{-Return hex string for word}`

(3) 9-17 `function HexL(L : LongInt) : string;`
`{-Return hex string for longint}`

(3) 9-17 `function HexPtr(P : Pointer) : string;`
`{-Return hex string for pointer}`

(3) 9-3 `function BinaryB(B : Byte) : string;`
`{-Return binary string for byte}`

OPSTRING

(3) 9-4
```
function BinaryW(W : Word) : string;
  {-Return binary string for word}
```
(3) 9-3
```
function BinaryL(L : LongInt) : string;
  {-Return binary string for longint}
```
(3) 9-22
```
function OctalB(B : Byte) : string;
  {-Return octal string for byte}
```
(3) 9-23
```
function OctalW(W : Word) : string;
  {-Return octal string for word}
```
(3) 9-22
```
function OctalL(L : LongInt) : string;
  {-Return octal string for longint}
```
(3) 9-27
```
function Str2Int(S : string; var I : Integer) : Boolean;
  {-Convert a string to an integer, returning true if successful}
```
(3) 9-27
```
function Str2Word(S : string; var I : Word) : Boolean;
  {-Convert a string to a word, returning true if successful}
```
(3) 9-27
```
function Str2Long(S : string; var I : LongInt) : Boolean;
  {-Convert a string to an longint, returning true if successful}
```
(3) 9-27
```
function Str2Real(S : string; var R : Float) : Boolean;
  {-Convert a string to a real, returning true if successful}
```
(3) 9-21
```
function Long2Str(L : LongInt) : string;
  {-Convert a longint/word/integer/byte/shortint to a string}
```
(3) 9-24
```
function Real2Str(R : Float; Width : Byte;
                  Places : ShortInt) : string;
  {-Convert a real to a string}
```
(3) 9-14
```
function Form(Mask : string; R : Float) : string;
  {-Returns a formatted string with digits from R merged into the
    Mask}
```
(3) 9-21
```
function LongIntForm(Mask : string; L : LongInt) : string;
  {-Returns a formatted string with digits from L merged into the
    Mask}
```
(3) 9-32
```
function UpCaseMac(Ch : Char) : Char;
  {-Uppercase character macro, no international character support}
```
(3) 9-20
```
function LoCaseMac(Ch : Char) : Char;
  {-Lowercase character macro, no international character support}
```
(3) 9-32
```
function Upcase(Ch : Char) : Char;
  {-Return uppercase of char, with international character support}
```
(3) 9-30
```
function StUpcase(S : string) : string;
  {-Convert lower case letters in string to uppercase, with intl
    chars}
```
(3) 9-20
```
function LoCase(Ch : Char) : Char;
  {-Return lowercase of char, with international character support}
```
(3) 9-26
```
function StLocase(S : string) : string;
  {-Convert upper case letters in string to lowercase, with intl
    chars}
```
(3) 9-7
```
function CharStr(Ch : Char; Len : Byte) : string;
  {-Return a string of length len filled with ch}
```
(3) 9-23
```
function PadCh(S : string; Ch : Char; Len : Byte) : string;
  {-Return a string right-padded to length len with ch}
```
(3) 9-23
```
function Pad(S : string; Len : Byte) : string;
  {-Return a string right-padded to length len with blanks}
```
(3) 9-20
```
function LeftPadCh(S : string; Ch : Char; Len : Byte) : string;
  {-Return a string left-padded to length len with ch}
```
(3) 9-19
```
function LeftPad(S : string; Len : Byte) : string;
  {-Return a string left-padded to length len with blanks}
```
(3) 9-30
```
function TrimLead(S : string) : string;
  {-Return a string with leading white space removed}
```
(3) 9-31
```
function TrimTrail(S : string) : string;
  {-Return a string with trailing white space removed}
```
(3) 9-30
```
function Trim(S : string) : string;
  {-Return a string with leading and trailing white space removed}
```
(3) 9-31
```
function TrimSpaces(S : string) : string;
  {-Return a string with leading and trailing spaces removed}
```
(3) 9-7
```
function CenterCh(S : string; Ch : Char; Width : Byte) : string;
  {-Return a string centered in a string of Ch with specified width}
```
(3) 9-6
```
function Center(S : string; Width : Byte) : string;
  {-Return a string centered in a blank string of specified width}
```

(3) 9-12
```
function Entab(S : string; TabSize : Byte) : string;
  {-Convert blanks in a string to tabs on spacing TabSize}
```
(3) 9-11
```
function Detab(S : string; TabSize : Byte) : string;
  {-Expand tabs in a string to blanks on spacing TabSize}
```
(3) 9-33
```
function WordCount(S : string; WordDelims : CharSet) : Byte;
  {-Given a set of word delimiters, return number of words in S}
```
(3) 9-33
```
function WordPosition(N : Byte; S : string;
                      WordDelims : CharSet) : Byte;
  {-Given a set of word delimiters, return start position of N'th
    word in S}
```
(3) 9-13
```
function ExtractWord(N : Byte; S : string;
                     WordDelims : CharSet) : string;
  {-Given a set of word delimiters, return the N'th word in S}
```
(3) 9-34
```
procedure WordWrap(InSt : string; var OutSt, Overlap : string;
                   Margin : Byte; PadToMargin : Boolean);
  {-Wrap InSt at Margin, storing the result in OutSt and the
    remainder in Overlap}
```
(3) 9-9
```
function CompString(S1, S2 : string) : CompareType;
  {-Return less, equal, greater if s1<s2, s1=s2, or s1>s2}
```
(3) 9-10
```
function CompUCString(S1, S2 : string) : CompareType;
  {-Compare two strings in a case insensitive manner}
```
(3) 9-10
```
function CompStruct(var S1, S2; Size : Word) : CompareType;
  {-Compare two fixed size structures}
```
(3) 9-25
```
function Search(var Buffer; BufLength : Word;
                var Match; MatLength : Word) : Word;
  {-Search through Buffer for Match. BufLength is length of range to
    search. MatLength is length of string to match. Returns number
    of bytes searched to find Match, $FFFF if not found.}
```
(3) 9-25
```
function SearchUC(var Buffer; BufLength : Word;
                  var Match; MatLength : Word) : Word;
  {-Search through Buffer for Match, CASE-INSENSITIVE.
    Otherwise same as Search.}
```
(3) 9-4
```
procedure BMMakeTable(MatchString : string; var BT : BTable);
  {-Build Boyer-Moore link table}
```
(3) 9-5
```
function BMSearch(var Buffer; BufLength : Word;
                  BT : BTable; MatchString : string) : Word;
  {-Search Buffer for MatchString. BufLength is length of range to
    search. Returns number of bytes searched to find MatchString,
    $FFFF if not found}
```
(3) 9-6
```
function BMSearchUC(var Buffer; BufLength : Word;
                    BT : BTable; MatchString : string) : Word;
  {-Search Buffer for MatchString, CASE-INSENSITIVE. Assumes
    MatchString is already in uppercase. Otherwise same as BMSearch}
```
(3) 9-26
```
function Soundex(S : string) : string;
  {-Return 4 character soundex of input string}
```
(3) 9-22
```
function MakeLetterSet(S : string) : LongInt;
  {-Return a bit-mapped long storing the individual letters contained
    in S}
```
(3) 9-8
```
function CompareLetterSets(Set1, Set2 : LongInt) : Word;
  {-Returns the sum of the values of the letters common to Set1
    and Set2}
```
(3) 9-29
```
function StringToHeap(S : string) : StringPtr;
  {-Allocate space for s and return pointer}
```
(3) 9-29
```
function StringFromHeap(P : StringPtr) : string;
  {-Return string at p}
```
(3) 9-12
```
procedure DisposeString(P : StringPtr);
  {-Deallocate space for string at p}
```
(3) 9-11
```
function DefaultExtension(Name : string; Ext : ExtStr) : string;
  {-Return a file name with a default extension attached}
```
(3) 9-13
```
function ForceExtension(Name : string; Ext : ExtStr) : string;
  {-Force the specified extension onto the file name}
```
(3) 9-18
```
function JustFilename(PathName : string) : string;
  {-Return just the filename and extension of a pathname}
```
(3) 9-19
```
function JustName(PathName : string) : string;
  {-Return just the name (no extension, no path) of a pathname}
```

(3) 9-18 `function JustExtension(Name : string) : ExtStr;`
`{-Return just the extension of a pathname}`

(3) 9-19 `function JustPathname(PathName : string) : string;`
`{-Return just the drive:directory portion of a pathname}`

(3) 9-3 `function AddBackSlash(DirName : string) : string;`
`{-Add a default backslash to a directory name}`

(3) 9-7 `function CleanPathName(PathName : string) : string;`
`{-Return a pathname cleaned up as DOS will do it}`

(3) 9-16 `function FullPathName(FName : string) : string;`
`{-Given FName (known to exist), return a full pathname}`

OpAsciiz Procedures and Functions

(3) 9-36 `function Asc2Str(var A : Asciiz) : string;`
`{-Convert Asciiz to Turbo string, truncating if longer than 255 chars}`

(3) 9-48 `procedure Str2Asc(S : string; var A : Asciiz);`
`{-Convert a Turbo string into an Asciiz}`

(3) 9-46 `function LenAsc(A : Asciiz) : Word;`
`{-Return the length of an Asciiz string}`

(3) 9-43 `procedure CopyAsc(var A : Asciiz; Start, Len : Word; var O : Asciiz);`
`{-Return a substring of a. Note start=0 for first char in a}`

(3) 9-44 `procedure DeleteAsc(var A : Asciiz; Start, Len : Word);`
`{-Delete len characters of a, starting at position start}`

(3) 9-42 `procedure ConcatAsc(var A, B, C : Asciiz);`
`{-Concatenate two Asciiz strings, returning a third}`

(3) 9-43 `procedure ConcatStr(var A : Asciiz; S : string; var C : Asciiz);`
`{-Concatenate a string to an asciiz, returning a new asciiz}`

(3) 9-45 `procedure InsertAsc(var Obj, A : Asciiz; Start : Word);`
`{-Insert asciiz obj at position start of a}`

(3) 9-45 `procedure InsertStr(Obj : string; var A : Asciiz; Start : Word);`
`{-Insert string obj at position start of a}`

(3) 9-47 `function PosStr(Obj : string; var A : Asciiz) : Word;`
`{-Return the position of the string obj in a, returning NotFound if not found}`

(3) 9-46 `function PosAsc(var Obja, A : Asciiz) : Word;`
`{-Return the position of obja in a, returning NotFound if not found}`

(3) 9-40 `function AscToHeap(var A : Asciiz) : AsciizPtr;`
`{-Put Asciiz on heap, returning a pointer, nil if insufficient memory}`

(3) 9-37 `procedure AscFromHeap(P : AsciizPtr; var A : Asciiz);`
`{-Return an Asciiz from the heap, empty if pointer is nil}`

(3) 9-44 `procedure DisposeAsc(P : AsciizPtr);`
`{-Dispose of heap space pointed to by P}`

(3) 9-47 `function ReadLnAsc(var F : Text; var A : Asciiz) : Boolean;`
`{-Read an Asciiz from text file, returning true if successful}`

(3) 9-48 `function WriteAsc(var F : Text; var A : Asciiz) : Boolean;`
`{-Write an Asciiz to text file, returning true if successful}`

(3) 9-41 `procedure AscUpcase(var A, B : Asciiz);`
`{-Uppercase the Asciiz in a, returning b}`

(3) 9-38 `procedure AscLocase(var A, B : Asciiz);`
`{-Lowercase the Asciiz in a, returning b}`

(3) 9-37 `procedure AscCharStr(Ch : Char; Len : Word; var A : Asciiz);`
`{-Return an Asciiz of length len filled with ch}`

(3) 9-39 `procedure AscPadCh(var A : Asciiz; Ch : Char; Len : Word;`
`var B : Asciiz);`
`{-Right-pad the Asciiz in a to length len with ch, returning b}`

(3) 9-39 `procedure AscPad(var A : Asciiz; Len : Word; var B : Asciiz);`
`{-Right-pad the Asciiz in a to length len with blanks, returning b}`

(3) 9-38 `procedure AscLeftPadCh(var A : Asciiz; Ch : Char; Len : Word;`
`var B : Asciiz);`
`{-Left-pad the Asciiz in a to length len with ch, returning b}`

(3) 9-38 `procedure AscLeftPad(var A : Asciiz; Len : Word; var B : Asciiz);`
`{-Left-pad the Asciiz in a to length len with blanks, returning b}`

```
(3) 9-41    procedure AscTrimLead(var A, B : Asciiz);
              {-Return an Asciiz with leading white space removed}
(3) 9-41    procedure AscTrimTrail(var A, B : Asciiz);
              {-Return an Asciiz with trailing white space removed}
(3) 9-40    procedure AscTrim(var A, B : Asciiz);
              {-Return an Asciiz with leading and trailing white space removed}
(3) 9-36    procedure AscCenterCh(var A : Asciiz; Ch : Char; Width : Word;
                                  var B : Asciiz);
              {-Return an Asciiz centered in an Asciiz of Ch with specified
                width}
(3) 9-36    procedure AscCenter(var A : Asciiz; Width : Word; var B : Asciiz);
              {-Return an Asciiz centered in an Asciiz of blanks with specified
                width}
(3) 9-42    function CompAsc(var a1, a2 : Asciiz) : AscCompareType;
              {-Return less, equal, greater if a1<a2, a1=a2, or a1>a2}
(3) 9-42    function CompUCAsc(var a1, a2 : Asciiz) : AscCompareType;
              {-Compare two Asciizs in a case insensitive manner}
```

OpStrDev Procedures and Functions

```
(3) 9-50    procedure ReadStr(var S : string);
              {-'Read' a string into S from our string buffer}
(3) 9-50    function ReturnStr : string;
              {-Return the contents of our string buffer}
```

OpDos Procedures and Functions

```
(3) 10-7    function DosVersion : Word;
              {-Returns the DOS version number. High byte has major version
                number, low byte has minor version number. Eg.,
                DOS 3.1 => $0301.}
(3) 10-20   function NumberOfDrives : Byte;
              {-Returns the number of logical drives}
(3) 10-23   procedure SelectDrive(Drive : Char);
              {-Selects the specified drive as default if possible}
(3) 10-7    function DefaultDrive : Char;
              {-Returns the default drive as an uppercase letter}
(3) 10-30   function ValidDrive(Drive : Char) : Boolean;
              {-Return True if the specified drive is valid}
(3) 10-14   function GetDiskInfo(Drive : Byte;
                                 var ClustersAvailable, TotalClusters,
                                 BytesPerSector,
                                 SectorsPerCluster : Word) : Boolean;
              {-Return technical info about the specified drive}
(3) 10-13   function GetDiskClass(Drive : Char;
                                  var SubstDriveChar : Char) : DiskClass;
              {-Return the disk class for the drive with the specified letter}
(3) 10-22   function ReadDiskSectors(Drive : Word; FirstSect : Longint;
                                     NumSects : Word; var Buf) : Boolean;
              {-Read absolute disk sectors.}
(3) 10-31   function WriteDiskSectors(Drive : Word; FirstSect : Longint;
                                      NumSects : Word; var Buf) : Boolean;
              {-Write absolute disk sectors.}
(3) 10-16   function GetFileMode(FName : string; var Attr : Word) : Byte;
              {-Returns a file's attribute in Attr and the DOS error code as the
                function result.}
(3) 10-13   function FlushDosBuffers(var F) : Boolean;
              {-Flush DOS's buffers for the specified file}
(3) 10-12   function FileHandlesLeft : Byte;
              {-Return the number of available file handles}
(3) 10-12   function FileHandlesOpen(CountDevices : Boolean) : Byte;
              {-Return the number of open files owned by a program}
(3) 10-24   procedure SetDta(DTAptr : Pointer);
              {-Set the DOS DTA to point to DTAptr}
(3) 10-14   procedure GetDta(var DTAptr : Pointer);
              {-Return the DOS DTA pointer}
```

(3) 10-21
```
function ParsePath(var InputPath, SearchPath,
                   LeadInPath : string) : Boolean;
  {-Takes a user entered path, trims blanks, and returns a valid
    global search path and a valid lead-in path.}
```
(3) 10-21
```
function PrintInstalled : Boolean;
  {-Returns True if PRINT.COM is installed}
```
(3) 10-28
```
function SubmitPrintFile(FileName : string) : Byte;
  {-This procedure submits a file to the PC DOS 3.0 or greater
    concurrent print utility.}
```
(3) 10-5
```
procedure CancelPrintFile(FileMask : string);
  {-Cancels the files matched by the file mask passed in FileMask.}
```
(3) 10-5
```
procedure CancelAllPrintFiles;
  {-Cancels all files in the print queue}
```
(3) 10-17
```
function GetPrintStatus(var QPtr : Pointer) : Byte;
  {-Halts printing, returns current error status, puts pointer to the
    filename queue in the QPtr variable. Filenames in the queue are
    64-byte ASCIIZ strings. The end of the queue is marked by a name
    starting with a null.}
```
(3) 10-8
```
procedure EndPrintStatus;
  {-Releases the spooler from the GetPrintStatus procedure.}
```
(3) 10-15
```
function GetEnvironmentString(SearchString : string) : string;
  {-Return a string from the environment}
```
(3) 10-24
```
function SetBlock(var Paragraphs : Word) : Boolean;
  {-Change size of DOS memory block allocated to this program}
```
(3) 10-9
```
procedure NoExecDosProc(ActionCode : ActionCodeType; Param : Word);
  {-Do-nothing ExecDosProc}
```
(3) 10-9
```
function ExecDos(Command : string; UseSecond : Boolean;
                 EDP : ExecDosProc) : Integer;
  {-Execute any DOS command}
```
(3) 10-29
```
function TextSeek(var F : Text; Target : LongInt) : Boolean;
  {-Do a Seek for a text file opened for input. Returns False in case
    of I/O error}
```
(3) 10-28
```
function TextFileSize(var F : Text) : LongInt;
  {-Return the size of text file F. Returns -1 in case of I/O error.}
```
(3) 10-29
```
function TextPos(var F : Text) : LongInt;
  {-Return the current position of the logical file pointer (that is,
    for the position of the physical file pointer, adjusted to
    account buffering). Returns -1 in case of I/O error.}
```
(3) 10-29
```
function TextFlush(var F : Text) : Boolean;
  {-Flush the buffer(s) for a text file. Returns False in case of
    I/O error.}
```
(3) 10-20
```
function OpenStdDev(var F : Text; StdHandle : Word) : Boolean;
  {-Assign the text file to a standard DOS device: 0, 1, 2, or 4}
```
(3) 10-18
```
function HandleIsConsole(Handle : Word) : Boolean;
  {-Return true if handle is the console device}
```
(3) 10-26
```
procedure SetRawMode(var F : Text; On : Boolean);
  {-Set "raw" mode on or off for the specified text file (must be a
    device)}
```
(3) 10-11
```
function ExistFile(FName : string) : Boolean;
  {-Return true if file is found}
```
(3) 10-11
```
function ExistOnPath(FName : string;
                     var FullName : string) : Boolean;
  {-Return true if fname is found in a) current directory,
    b) program's directory (DOS 3.X only), or c) any DOS path
    directory and return full path name to file}
```
(3) 10-18
```
function IsDirectory(FName : String) : Boolean;
  {-Return true if FName is a directory}
```
(3) 10-23
```
function SameFile(FilePath1,FilePath2 : String;
                  var ErrorCode : Word) : Boolean;
  {-Return true if FilePath1 and FilePath2 refer to the same
    physical file. Error codes: 0 - Success (no error),
    1 - Invalid FilePath}
```
(3) 10-6
```
function CopyFile(SrcPath, DestPath : String;
                  Buffer : Pointer;
                  BufferSize : Word) : Word;
  {-Copy the file specified by SrcPath into DestPath}
```

(3) 10-30
```
function TimeMs : LongInt;
  {-Return time of day in milliseconds since midnight}
```
(3) 10-19
```
procedure MasterEnv(var Env : EnvRec);
  {-Return master environment record}
```
(3) 10-6
```
procedure CurrentEnv(var Env : EnvRec);
  {-Return current environment record}
```
(3) 10-20
```
procedure ParentEnv(var Env : EnvRec);
  {-Return environment record of program's parent}
```
(3) 10-19
```
procedure NewEnv(var Env : EnvRec; Size : Word);
  {-Allocate a new environment on the heap}
```
(3) 10-7
```
procedure DisposeEnv(var Env : EnvRec);
  {-Deallocate an environment previously allocated on heap}
```
(3) 10-24
```
procedure SetCurrentEnv(Env : EnvRec);
  {-Specify a different environment for the current program}
```
(3) 10-5
```
procedure CopyEnv(Src, Dest : EnvRec);
  {-Copy contents of Src environment to Dest environment}
```
(3) 10-8
```
function EnvFree(Env : EnvRec) : Word;
  {-Return bytes free in environment}
```
(3) 10-15
```
function GetEnvStr(Env : EnvRec; Search : string) : string;
  {-Return a string from the environment}
```
(3) 10-25
```
function SetEnvStr(Env : EnvRec; Search, Value : string) : Boolean;
  {-Set environment string, returning true if successful}
```
(3) 10-8
```
procedure DumpEnv(Env : EnvRec);
  {-Dump the environment to the screen}
```
(3) 10-17
```
function GetProgramStr(Env : EnvRec) : string;
  {-Return the name of the program that owns Env, '' if DOS < 3.0
    or unknown}
```
(3) 10-26
```
function SetProgramStr(Env : EnvRec; Path : string) : Boolean;
  {-Add a program name to the end of an environment if sufficient
    space}
```
(3) 10-27
```
function ShellWithPrompt(Prompt : string;
                         EDP : ExecDosProc) : Integer;
  {-Shell to DOS with a new prompt}
```

OpExec Procedures and Functions

(3) 10-41
```
procedure SetSwapMsgOn(On : Boolean);
  {-Turns Swap message on or off}
```
(3) 10-39
```
procedure ExecWithSwap(Path, CmdLine : String);
  {-DOS shell with swapping to EMS or disk}
```
(3) 10-40
```
function PrepareExecWithSwap(LastToSave : Pointer;
                             SwapFileName : PathStr) : Boolean;
  {-Set up for a shell with swapping, returning true if successful}
```
(3) 10-41
```
procedure RemoveExecWithSwap;
  {-Deallocate EMS space or erase swap file, whichever was allocated}
```
(3) 10-36
```
function ExecDosSwap(Command : string; UseSecond : Boolean;
                     EDP : Pointer; SwapFName : PathStr) : Integer;
  {-Execute any DOS command}
```

OpEnhKbd Procedures and Functions

(3) 10-45
```
procedure RestoreKbdVectors;
  {-Restores original vectors for INT's $09 and $16}
```
(3) 10-45
```
procedure InitKbdVectors;
  {-Save and setup interrupt vectors. This routine should be called
    only if RestoreKbdVectors has been called first!!}
```

OpInt Procedures and Functions

(3) 11-8
```
procedure InterruptsOn;
  {-Turn interrupts on}
```
(3) 11-8
```
procedure InterruptsOff;
  {-Turn interrupts off}
```

OPINT

(3) 11-10
```
procedure SendEOI;
  {-Send an End Of Interrupt command to the Programmable Interrupt
    Controller}
```
(3) 11-9
```
procedure IntReturn(var IntRegs : IntRegisters);
  {-Return from interrupt. Needed only if stack has been changed.}
```
(3) 11-6
```
procedure ChainInt(var Regs : IntRegisters; JumpAddr : Pointer);
  {-Restores stack, registers from Regs and 'jumps' to JumpAddr}
```
(3) 11-11
```
procedure SwapStackAndCallNear(Routine : Word; SP : Pointer;
                               var Regs : IntRegisters);
  {-Switches to stack designated by SP and calls Routine with Regs
    as a parameter. The Routine must be a NEAR call from the current
    ISR.}
```
(3) 11-11
```
procedure SwapStackAndCall(Routine, SP : Pointer;
                           var Regs : IntRegisters);
  {-Switches to stack designated by SP and calls Routine with Regs
    as a parameter. The Routine must be a FAR call from the current
    ISR.}
```
(3) 11-7
```
procedure EmulateInt(var Regs : IntRegisters; IntAddr : Pointer);
  {-Emulates an interrupt by filling the CPU registers with the
    values in Regs, clearing interrupts, pushing the flags, and
    calling far to IntAddr.}
```
(3) 11-8
```
function InitVector(IntNumber, Handle : Byte;
                    UserRoutine : Pointer) : Boolean;
  {-Sets up an interrupt service routine}
```
(3) 11-10
```
procedure RestoreVector(Handle : Byte);
  {-Restores an interrupt vector to its original value}
```
(3) 11-9
```
procedure RestoreAllVectors;
  {-Restores all captured interrupt vectors.}
```
(3) 11-6
```
function AllocateStack(var P : Pointer;
                       SizeInBytes : Word) : Boolean;
  {-Allocates a stack of size SizeInBytes}
```
(3) 11-7
```
procedure DeallocateStack(P : Pointer);
  {-Frees an allocated stack}
```

OpTsr Procedures and Functions

(3) 11-30
```
function ParagraphsToKeep : Word;
  {-Returns # of paragraphs currently used for code, data, stack,
    and heap.}
```
(3) 11-29
```
function MaxParagraphsToKeep : Word;
  {-Maximum number of paragraphs to keep when going resident}
```
(3) 11-33
```
procedure StayRes(ParasToKeep : Word; ExitCode : Byte);
  {-Terminate and stay resident, freeing up all our memory except
    ParasToKeep. Returns only if unable to go resident.}
```
(3) 11-31
```
procedure PopupsOn;
  {-Turns popups on}
```
(3) 11-31
```
procedure PopupsOff;
  {-Turns popups off}
```
(3) 11-25
```
function DefinePopProc(var Handle : Byte; Routine : PopupProc;
                       StackPtr : Pointer) : Boolean;
  {-Defines a routine that can be triggered by calling SetPopTicker.
    Returns false if no more Popup handles are available.}
```
(3) 11-26
```
procedure DeletePopProc(Handle : Byte);
  {-Deletes a popup routine defined with DefinePopProc.  Its handle
    may then be used for other popups.}
```
(3) 11-25
```
function DefinePop(HotKey : Word; Routine : PopupProc;
                   StackPtr : Pointer; DosWait : Boolean) : Boolean;
  {-Defines a popup routine associated with a particular hotkey.
    Returns False if no more Popup handles are available.}
```
(3) 11-26
```
procedure DeletePop(HotKey : Word);
  {-Deletes the popup routine associated with HotKey.  Its handle
    may then be used for other popups.}
```
(3) 11-24
```
function ChangeHotKey(OldHotKey, NewHotKey : Word) : Boolean;
  {-Change a popup's hotkey from OldHotKey to NewHotKey}
```

(3) 11-24
```
function AddHotKey(PrimaryHotKey, SecondaryHotKey : Word) : Boolean;
  {-Add a second hot key for the popup with the specified primary
    hot key}
```
(3) 11-32
```
procedure SetPopTicker(Handle : Byte; TimeOut : Word);
  {-Sets the pop ticker for the indicated procedure, clears
    interrupts, then returns.  No range checking is performed.}
```
(3) 11-31
```
procedure RemovePops;
  {-Removes all popup routines from the system and releases all
    associated interrupt vectors}
```
(3) 11-28
```
procedure InitPops;
  {-Must be called before any other popup specific routines}
```
(3) 11-27
```
function DisableTSR : Boolean;
  {-Disable TSR by restoring interrupt vectors and releasing memory.
    This does *not* halt the program. Returns False if it's not
    safe.}
```
(3) 11-32
```
function SafeToDisable : Boolean;
  {-Returns True if it is safe to disable the TSR}
```
(3) 11-28
```
procedure InstallModule(var ModuleName : string;
                            CmdEntryRoutine : Pointer);
  {-Installs this program as a resident module that can be located
    and accessed by other programs.}
```
(3) 11-33
```
procedure UninstallModule;
  {-Uninstalls the module from the linked list of modules}
```
(3) 11-30
```
function ModulePtrByName(var ModuleName : string) : IfcPtr;
  {-Returns a pointer to the IfcRecord for the module named
    ModuleName or Nil}
```
(3) 11-29
```
function ModuleInstalled(var ModuleName : string) : Boolean;
  {-Returns true if ModuleName is installed.}
```
(3) 11-27
```
function DosBusyFlag : Byte;
  {-Returns current value of DOS busy flag}
```
(3) 11-27
```
function DosCriticalFlag : Byte;
  {-Returns current value of DOS critical flag}
```
(3) 11-28
```
function GetPSP : Word;
  {-Returns current PSP}
```
(3) 11-32
```
procedure SetPSP(PSP : Word);
  {-Sets current PSP}
```

OpSwap Procedures and Functions

(3) 11-57
```
procedure SetPopTicker(TimeOut : Word);
  {-Sets the pop ticker for the CmdEntry procedure, clears
    interrupts, then returns.}
```

OpSwap1 Procedures and Functions

(3) 11-48
```
function AllocateStack(var P : Pointer;
                           SizeInBytes : Word) : Boolean;
  {-Allocates a stack of size SizeInBytes}
```
(3) 11-48
```
procedure DeallocateStack(P : Pointer);
  {-Frees an allocated stack}
```
(3) 11-49
```
procedure PopupsOn;
  {-Turns popups on}
```
(3) 11-49
```
procedure PopupsOff;
  {-Turns popups off}
```
(3) 11-48
```
function ChangeHotKey(OldHotKey, NewHotKey : Word) : Boolean;
  {-Change a popup's hotkey from OldHotKey to NewHotKey}
```
(3) 11-48
```
function AddHotKey(PrimaryHotKey, SecondaryHotKey : Word) : Boolean;
  {-Add a second hot key for the popup with the specified primary
    hot key}
```
(3) 11-50
```
function DefinePop(HotKey : Word; Routine : PopupProc;
                       StackPtr : Pointer) : Boolean;
  {-Defines a popup routine associated with a particular hotkey.
    Returns False if no more Popup handles are available.}
```
(3) 11-49
```
procedure RestoreAllVectors;
  {-Restores all captured interrupt vectors}
```

OPSWAP1

(3) 11-55 `function WillSwapUseEms(ParasToKeep : Word) : Boolean;`
`{-Returns True if OpSwap will use EMS memory for swapping}`

(3) 11-49 `function ParagraphsToKeep : Word;`
`{-Returns # of paragraphs currently used for code, data, stack, and heap}`

(3) 11-52 `function MaxParagraphsToKeep : Word;`
`{-Maximum number of paragraphs to keep when going resident}`

(3) 11-55 `procedure StayResSwap(ParasToKeep : Word; ExitCode : Byte; SwapFileName1, SwapFileName2 : String; UseSwapping : Boolean);`
`{-Terminate and stay resident. If UseSwapping is False, then the TSR does not use the swapping techniques and behaves like a normal TSR written with OpTSR (or TpTSR)}`

(3) 11-48 `function DisableTSR : Boolean;`
`{-Disable TSR by restoring interrupt vectors and releasing memory. This does *not* halt the program. Returns false if it's not safe.}`

(3) 11-49 `function SafeToDisable : Boolean;`
`{-Returns True if it is safe to disable the TSR}`

(3) 11-51 `procedure DisablePopup;`
`{-An external interface procedure that attempts to disable the popup and reports the results in the UserData field of the IfcRecord. UserData is set to 1 on success.}`

(3) 11-52 `procedure InstallModule(var ModName : string; CmdEntryPtr : PopupProc);`
`{-Installs this program as a resident module that can be located and accessed by other programs.}`

(3) 11-49 `procedure UninstallModule;`
`{-Uninstalls the module from the linked list of modules.}`

(3) 11-53 `procedure SetSwapMsgAttr(Attr : Byte);`
`{-Sets the attribute used by Swap manager for swap message}`

(3) 11-53 `procedure SetSwapMsgOn(On : Boolean);`
`{-Turns Swap message on or off}`

(3) 11-53 `procedure SetSwapMsgRow(Row : Byte);`
`{-Sets the row where swapping messages are to appear}`

(3) 11-54 `procedure SetUserExitProc(UserExitProc : PopupProc);`
`{-Sets a pointer to a far procedure that is to be called before disabling the TSR}`

(3) 11-48 `function ModulePtrByName(var ModuleName : string) : IfcPtr;`
`{-Returns a pointer to the IfcRecord for the module named ModuleName or Nil.}`

(3) 11-48 `function ModuleInstalled(var ModuleName : string) : Boolean;`
`{-Returns true if ModuleName is installed.}`

(3) 11-48 `procedure EmulateInt(var Regs : IntRegisters; IntAddr : Pointer);`
`{-Emulates an interrupt by filling the CPU registers with the values in Regs, clearing interrupts, pushing the flags, and calling far to IntAddr.}`

(3) 11-48 `function DosBusyFlag : Byte;`
`{-Returns current value of DOS busy flag}`

(3) 11-48 `procedure InterruptsOn;`
`{-Turn interrupts on}`

(3) 11-48 `procedure InterruptsOff;`
`{-Turn interrupts off}`

Op8087 Procedures and Functions

(3) 11-63 `procedure Save8087(var SaveBuf : SaveBuffer8087);`
`{-Saves the 80x87 registers in the save buffer.}`

(3) 11-63 `procedure Restore8087(var SaveBuf : SaveBuffer8087);`
`{-Restores the 80x87 registers from the save buffer.}`

(3) 11-62 `function Installed8087 : Boolean;`
`{-Returns true if an 80x87 coprocessor is installed.}`

(3) 11-61 `procedure Exceptions8087(On : Boolean);`
`{-Turn exception interrupts on or off}`

(3) 11-61 `function Error8087 : Word;`
`{-Return (and clear) the error status of the 80x87}`

(3) 11-62 procedure Reinit8087;
{-Reinitialize the 8087 chip}

(3) 11-62 procedure ReinitEmulator;
{-Reinitialize the emulator}

(3) 11-63 function UseEmulator : Boolean;
{-Return True if ReinitEmulator should be used rather than Reinit8087}

OpEms Procedures and Functions

(3) 12-6 function EmsInstalled : Boolean;
{-Returns true if the EMM is installed.}

(3) 12-5 function EmmSigFound : Boolean;
{-Returns True if EMM signature found}

(3) 12-9 function EmsStatusOK : Boolean;
{-Returns true if the EMM reports its status as being OK.}

(3) 12-8 function EmsPagesAvail : Word;
{-Returns the number of available pages from the expanded memory manager, or EmsErrorCode in case of error.}

(3) 12-9 function EmsTotalPages : Word;
{-Returns total number of pages of EMS memory, including allocated pages, or EmsErrorCode in case of error.}

(3) 12-7 function EmsPageFramePtr : Pointer;
{-Returns the page frame base address as a pointer to the page frame.}

(3) 12-4 function AllocateEmsPages(NumPages : Word) : Word;
{-Allocates the indicated number of pages and returns a handle. Returns EmsErrorCode in case of error.}

(3) 12-10 function MapEmsPage(Handle, LogicalPage : Word;
PhysicalPage : Byte) : Boolean;
{-Maps the specified LogicalPage associated with Handle into PhysicalPage (0-3). Returns true if successful.}

(3) 12-5 function DeallocateEmsHandle(Handle : Word) : Boolean;
{-Deallocates the indicated handle and the memory associated with it.}

(3) 12-10 function EmsVersion : Byte;
{-Returns a BCD version number of the EMM handle. To check for version 3.2 or greater for example, use: 'if EmsVersion >= $32 then'. Returns 0 in case of error.}

(3) 12-11 function SaveEmsContext(Handle : Word) : Boolean;
{-Saves the EMM context for resident programs. The handle passed must have been allocated with a call to AllocateEmsPages. Returns true if successful.}

(3) 12-11 function RestoreEmsContext(Handle : Word) : Boolean;
{-Restores the mapping context of the EMM driver for the handle specified. The handle should the same one used in a prior call to SaveEmsContext. Returns true if successful.}

(3) 12-6 function EmsActiveHandles : Word;
{-Returns the number of active EMS handles, EmsErrorCode in case of error}

(3) 12-8 function EmsPagesOwned(Handle : Word) : Word;
{-Returns the number of pages owned by Handle, or EmsErrorCode in case of error.}

OpExtMem Procedures and Functions

(3) 12-16 function PtrToHuge(P : Pointer) : HugePtr;
{-Converts a pointer to a huge pointer. The pointer must be in the range $0:$0 to $FFFF:$000F.}

(3) 12-15 procedure MoveExtMem(Src, Dest : HugePtr; WCount : Word);
{-Move WCount words of extended memory from Src to Dest.}

(3) 12-13 function ExtMemTotal : LongInt;
{-Returns total number of bytes of extended memory in the system.}

(3) 12-13 function ExtMemAvail : Word;
{-Returns the number of KB of extended memory that is actually free.}

OPEXTMEM

(3) 12-14
```
function GetExtMem(N : Word) : HugePtr;
  {-Allocate N Kb of extended memory.  Returns Nil if insufficient
    memory.}
```
(3) 12-14
```
function FreeExtMem(P : HugePtr) : Boolean;
  {-Deallocates extended memory which has been allocated with
    GetExtMem.}
```
(3) 12-15
```
procedure MarkExtMem(var P : HugePtr);
  {-Store the location of the next extended memory block in P}
```
(3) 12-16
```
procedure ReleaseExtMem(P : HugePtr);
  {-Release all extended memory above P. P *must* have been set by
    MarkExtMem. It cannot be a huge pointer returned by GetExtMem.}
```

OpMacro Procedures and Functions

(3) 13-7
```
function DefineMacro(Key : Word; var Macro) : Boolean;
  {-Define a new macro or delete an existing one.}
```
(3) 13-9
```
function MacroAddress(Key : Word) : Pointer;
  {-Returns a pointer to the macro for a key, or nil if key is not
    defined.}
```
(3) 13-9
```
function MacroCount : Word;
  {-Returns the number of macros that have been defined}
```
(3) 13-5
```
function AllocateMacro(Key : Word; var Macro;
                       var Name : string) : Byte;
  {-Allocates a macro on the heap, and returns one of the following
    values: 0 : Macro allocated; 1 : Macro table full; 2 : Out of
    heap space; 3 : Macro was blank }
```
(3) 13-6
```
function DeallocateMacro(Key : Word) : Boolean;
  {-If the macro exists, this routine erases it and deallocates its
    memory. Returns False if the macro was not found.}
```
(3) 13-5
```
function AssignMacroName(Key : Word; var Name : string) : Boolean;
  {-Assign a Name to a macro, deleting the existing name assigned to
    the macro, if any. Returns false if insufficient memory exists
    or macro not found.}
```
(3) 13-11
```
function ReadMacroFile(FName : string; Merge : Boolean) : Byte;
  {-Read a macro file. Returns 0 if successful, $FF if not a valid
    macro file, $FE if we ran out of memory, else IOresult. If Merge
    is false, any existing macros are deleted before the file is
    read.}
```
(3) 13-14
```
function WriteMacroFile(FName : string) : Byte;
  {-Write a macro file. Returns 0 if successful, else IOresult.}
```
(3) 13-6
```
procedure ClearMacros;
  {-Clears out the current macros.}
```
(3) 13-10
```
procedure MacrosOn;
  {-Turns macro processing on}
```
(3) 13-10
```
procedure MacrosOff;
  {-Turns macro processing off}
```
(3) 13-10
```
procedure MacroRecordingOn;
  {-Turns macro recording on}
```
(3) 13-9
```
procedure MacroRecordingOff;
  {-Turns macro recording off}
```
(3) 13-13
```
procedure StartMacro(Macro : Pointer);
  {-Starts execution of the specified macro. If a macro is already in
    progress it will immediately be replaced by the macro passed.}
```
(3) 13-13
```
procedure StartMacroByKey(Key : Word);
  {-Starts execution of the macro attached to key, if any. If a macro
    is already in progress it will immediately be replaced by the
    macro passed.}
```
(3) 13-11
```
procedure RemoveMacros;
  {-Restores all vectors associated with macros.}
```
(3) 13-12
```
procedure SetEndOfMacroProc(P : Pointer);
  {-Set pointer to procedure to call when end of macro reached}
```
(3) 13-7
```
function FindMacroIndex(Key : Word) : Word;
  {-Find the array index for the macro assigned to Key. Returns 0 if
    not found.}
```

(3) 13-8
```
function InitMacros : Boolean;
  {-Initializes macros. After this routine is called, no macros will
    be defined and both macro processing and macro recording will be
    turned off. Returns false if INT $16 cannot be taken over.}
```
(3) 13-6
```
function CharToMacro(MacroChar : Char) : Word;
  {-Converts a character to a keyboard code/char representation.
    This table holds all scan code values for ASCII characters from
    0 to 63. All alpha characters can be mapped to this table by
    subtracting a number divisible by 32 that would put that alpha
    in the range of 0..63.}
```
(3) 13-13
```
procedure StringToMacro(S : string; MRP : MacroRecPtr;
                        MaxKeys : Word);
  {-Initialize MRP^ from a string}
```
(3) 13-14
```
procedure StringToScrapMacro(S : string);
  {-Initialize ScrapMacro from a string}
```
(3) 13-8
```
procedure KeyToString(Key : Word; var S : string;
                      var Special : Boolean);
  {-Returns a string (S) representing a Key. Special is set to False
    if a simple character is being returned.}
```

OpMacEd Procedures and Functions

(3) 13-16
```
function EditKeys(Key : Word; var Macro : MacroRec;
                  MinCol, MinRow, MaxCol, MaxRow,
                  Dim, Bright : Byte;
                  var Modified : Boolean) : MacroRecPtr;
  {-Edit the Macro associated with Key.}
```

OpReplay Procedures and Functions

(3) 13-21
```
procedure StartMacro(Macro : MacroRecPtr);
  {-Starts execution of the specified macro}
```
(3) 13-19
```
function CharToMacro(MacroChar : Char) : Word;
  {-Converts a character to a keyboard code/char representation}
```
(3) 13-22
```
procedure StringToScrapMacro(S : string);
  {-Initialize ScrapMacro from a string}
```
(3) 13-22
```
procedure StringToMacro(S : string; MRP : MacroRecPtr;
                        MaxKeys : Word);
  {-Initialize MRP^ from a string}
```
(3) 13-20
```
procedure Int16;
  {-The macro engine used to replay keystrokes. It is the user's
    responsibility to install and uninstall this interrupt handler.}
```
(3) 13-21
```
procedure SetEndOfMacroProc(P : Pointer);
  {-Set pointer to procedure to call when end of macro reached}
```
(3) 13-19
```
procedure InitScrapMacroPtr;
  {-Initialize ScrapMacroPtr}
```

OpBCD Procedures and Functions

(3) 14-11
```
procedure RealToBCD(R : Real; var B : BCD);
  {-Convert a real to a BCD}
```
(3) 14-9
```
procedure LongintToBCD(L : LongInt; var B : BCD);
  {-Convert a Longint to a BCD}
```
(3) 14-5
```
function BCDtoReal(B : BCD) : Real;
  {-Convert a BCD to a real}
```
(3) 14-13
```
function StrBCD(B : BCD; Width, Places : Byte) : string;
  {-Return a BCD as a string}
```
(3) 14-13
```
function StrExpBCD(B : BCD; Width : Byte) : string;
  {-Return B as a string in exponential format}
```
(3) 14-15
```
procedure ValBCD(S : string; var B : BCD; var Code : Word);
  {-Convert a string to a BCD}
```
(3) 14-4
```
procedure AbsBCD(B1 : BCD; var B2 : BCD);
  {-Returns absolute value of B1 in B2}
```
(3) 14-7
```
procedure FracBCD(B1 : BCD; var B2 : BCD);
  {-Returns the fractional part of B1 in B2}
```

OPBCD

(3) 14-8 `procedure IntBCD(B1 : BCD; var B2 : BCD);`
`{-Returns the integer part of B1 in B2}`

(3) 14-11 `function RoundBCD(B1 : BCD) : LongInt;`
`{-Returns the value of B1 rounded to the nearest long integer}`

(3) 14-14 `function TruncBCD(B1 : BCD) : LongInt;`
`{-Returns the greatest long integer less than or equal to B1}`

(3) 14-7 `function Form(Mask : string; B : BCD) : string;`
`{-Returns a formatted string with digits from B merged into the Mask}`

(3) 14-4 `procedure AddBCD(B1, B2 : BCD; var B3 : BCD);`
`{-Add B1 to B2 and put result in B3}`

(3) 14-14 `procedure SubBCD(B1, B2 : BCD; var B3 : BCD);`
`{-Subtract B2 from B1 and put result in B3}`

(3) 14-10 `procedure MultBCD(B1, B2 : BCD; var B3 : BCD);`
`{-Multiply B1 by B2 and put result in B3}`

(3) 14-5 `procedure DivBCD(B1, B2 : BCD; var B3 : BCD);`
`{-Divide B1 by B2 and put result in B3}`

(3) 14-6 `function EqualBCD(B1, B2 : BCD) : Boolean;`
`{-Returns true if B1 = B2}`

(3) 14-10 `function NotEqualBCD(B1, B2 : BCD) : Boolean;`
`{-Returns true if B1 <> B2}`

(3) 14-7 `function GreaterBCD(B1, B2 : BCD) : Boolean;`
`{-Returns true if B1 > B2}`

(3) 14-8 `function GreaterEqualBCD(B1, B2 : BCD) : Boolean;`
`{-Returns true if B1 >= B2}`

(3) 14-8 `function LessBCD(B1, B2 : BCD) : Boolean;`
`{-Returns true if B1 < B2}`

(3) 14-9 `function LessEqualBCD(B1, B2 : BCD) : Boolean;`
`{-Returns true if B1 <= B2}`

(3) 14-4 `procedure ArcTanBCD(B1 : BCD; var B2 : BCD);`
`{-Returns arc tangent of B1 in B2}`

(3) 14-5 `procedure CosBCD(B1 : BCD; var B2 : BCD);`
`{-Returns cosine of B1 in B2}`

(3) 14-6 `procedure ExpBCD(B1 : BCD; var B2 : BCD);`
`{-Returns the exponential of B1 in B2}`

(3) 14-9 `procedure LnBCD(B1 : BCD; var B2 : BCD);`
`{-Returns the natural log of B1 in B2}`

(3) 14-12 `procedure SinBCD(B1 : BCD; var B2 : BCD);`
`{-Returns the sine of B1 in B2}`

(3) 14-12 `procedure SqrBCD(B1 : BCD; var B2 : BCD);`
`{-Returns the square of B1 in B2}`

(3) 14-12 `procedure SqrtBCD(B1 : BCD; var B2 : BCD);`
`{-Returns the square root of B1 in B2}`

OpClone Cloner

(3) 14-21 `constructor Init(FName : string; DUT : DateUpdateType);`
`{-Open file for cloning}`

(3) 14-21 `constructor InitCustom(FName : string; DUT : DateUpdateType; BufSize : Word);`
`{-Open file for cloning}`

(3) 14-22 `constructor InitAndFind(FName : string; DUT : DateUpdateType; var IdSize : Word);`
`{-Open file and find ID. Uses FindDefaultsEnd}`

(3) 14-19 `destructor Done; virtual;`
`{-Close clone file, adjust time stamp, and deallocate buffer}`

(3) 14-19 `function FindDefaultsEnd(var ID; IdSize : Word; Skip : LongInt) : Boolean;`
`{-Find the ID in the clone file, searching from the end backward}`

(3) 14-19 `function FindDefaultsStart(var ID; IdSize : Word; Skip : LongInt) : Boolean;`
`{-Find the ID in the clone file, searching from the start forward}`

(3) 14-20 `function GetLastError : Word;`
`{-Get the code for the last error}`

(3) 14-21 `function GetPos : LongInt;`
`{-Get position where an ID string was found (offset returned is the start of the ID string)}`

(3) 14-23 `procedure LoadDefaults(FileOfs : LongInt; var Defaults; Bytes : Word);`
`{-Seek to position FileOfs and read defaults there}`

(3) 14-24 `procedure StoreDefaults(FileOfs : LongInt; var Defaults; Bytes : Word);`
`{-Seek to position FileOfs and store defaults there}`

OpErrHan Procedures and Functions

(3) 14-27 `procedure ErrorRecovery(On : Boolean; UserRoutine : Pointer);`
`{-Turn error recovery on and off, and attach to a user error routine}`

OpInline Procedures and Functions

(3) 14-43 `function SetJump(var JumpDest : JumpRecord) : Word;`
`{-Save current SP, BP, and a jump destination}`

(3) 14-37 `procedure LongJump(var JumpDest : JumpRecord; Result : Word);`
`{-Restore SP, BP, and jump to JumpDest.JmpPt}`

(3) 14-34 `procedure FarCall(ProcAddr : Pointer);`
`{-ProcAddr is the address of a routine to be called far. Can be used to implement jump tables if procedures take no parameters.}`

(3) 14-41 `procedure NearCall(ProcOfs : Word);`
`{-ProcOfs is the offset of a routine to be called near.}`

(3) 14-36 `procedure JumpToOldIsr(OldIsr : Pointer);`
`{-Jump to previous ISR from an interrupt procedure.}`

(3) 14-39 `function MakeLongInt(H, L : Word) : LongInt;`
`{-Constructs a LongInt from two Words}`

(3) 14-38 `function MakeInteger(H, L : Byte): Integer;`
`{-Constructs an integer from two bytes}`

(3) 14-39 `function MakeWord(H, L : Byte) : Word;`
`{-Constructs a word from two bytes}`

(3) 14-30 `function Array2Str(var A; Len : Byte) : string;`
`{-Convert an array of char to a string}`

(3) 14-31 `procedure CallOldIsr(OldIsr : Pointer);`
`{-Call previous ISR from an interrupt procedure. Destroys BX.}`

(3) 14-42 `procedure Reboot;`
`{-Reboot the machine}`

(3) 14-36 `function HiWord(L : LongInt) : Word;`
`{-Return high-order word of L}`

(3) 14-38 `function LoWord(L : LongInt) : Word;`
`{-Return low-order word of L}`

(3) 14-44 `function SwapNibble(B : Byte) : Byte;`
`{-Swap the high and low nibbles of B: SwapNibble($F0) returns $0F.}`

(3) 14-44 `function SwapWord(L : LongInt) : LongInt;`
`{-Swap low- and high-order words of L}`

(3) 14-41 `function Normalized(P : Pointer) : Pointer;`
`{-Return P as a normalized pointer}`

(3) 14-43 `procedure SetFlag(var Flags : Word; FlagMask : Word);`
`{-Set the bit(s) specified by FlagMask in Flags}`

(3) 14-31 `procedure ClearFlag(var Flags : Word; FlagMask : Word);`
`{-Clear the bit(s) specified by FlagMask in Flags}`

(3) 14-35 `function FlagIsSet(Flags, FlagMask : Word) : Boolean;`
`{-Returns True if the bit specified by FlagMask is set in Flags}`

(3) 14-42 `procedure SetByteFlag(var Flags : Byte; FlagMask : Byte);`
`{-Set the bit(s) specified by FlagMask in Flags}`

(3) 14-31 `procedure ClearByteFlag(var Flags : Byte; FlagMask : Byte);`
`{-Clear the bit(s) specified by FlagMask in Flags}`

(3) 14-30 `function ByteFlagIsSet(Flags, FlagMask : Byte) : Boolean;`
`{-Returns True if the bit specified by FlagMask is set in Flags}`

(3) 14-44 `procedure SetLongFlag(var Flags : LongInt; FlagMask : LongInt);`
`{-Set the bit(s) specified by FlagMask in Flags}`

(3) 14-32
```
procedure ClearLongFlag(var Flags : LongInt; FlagMask : LongInt);
  {-Clear the bit(s) specified by FlagMask in Flags}
```
(3) 14-37
```
function LongFlagIsSet(Flags, FlagMask : LongInt) : Boolean;
  {-Returns True if the bit specified by FlagMask is set in Flags}
```
(3) 14-32
```
procedure ExchangeBytes(var I, J : Byte);
  {-Exchange bytes I and J}
```
(3) 14-33
```
procedure ExchangeWords(var I, J : Word);
  {-Exchange words I and J}
```
(3) 14-32
```
procedure ExchangeLongInts(var I, J : LongInt);
  {-Exchange LongInts I and J}
```
(3) 14-33
```
procedure ExchangeStructs(var I, J; Size : Word);
  {-Exchange structures I and J. Useful in sorts}
```
(3) 14-40
```
function MinWord(A, B : Word) : Word;
  {-Returns the smaller of A and B}
```
(3) 14-40
```
function MaxWord(A, B : Word) : Word;
  {-Returns the greater of A and B}
```
(3) 14-40
```
function MinLong(A, B : LongInt) : LongInt;
  {-Returns the smaller of A and B}
```
(3) 14-39
```
function MaxLong(A, B : LongInt) : LongInt;
  {-Returns the greater of A and B}
```
(3) 14-35
```
procedure FillWord(var Dest; Count, Filler : Word);
  {-Fill memory starting at Dest with Count instances of Filler}
```
(3) 14-34
```
procedure FillStruct(var Dest; Count : Word; var Filler;
                     FillerSize : Word);
  {-Fill memory starting at Dest with Count instances of Filler}
```
(3) 14-29
```
function AddWordToPtr(P : Pointer; W : Word) : Pointer;
  {-Add a Word to a pointer. No normalization or wrap checking
    performed}
```
(3) 14-42
```
function PtrToLong(P : Pointer) : LongInt;
  {-Convert pointer, in range $0:$0 to $FFFF:$000F, to LongInt}
```
(3) 14-38
```
function LongToPtr(L : LongInt) : Pointer;
  {-Return LongInt L as a normalized pointer}
```
(3) 14-41
```
function PtrDiff(P1, P2 : Pointer) : LongInt;
  {-Return the number of bytes between P1^ and P2^}
```
(3) 14-29
```
function AddLongToPtr(P : Pointer; L : LongInt) : Pointer;
  {-Add a LongInt to a pointer, returning a normalized pointer}
```

OpSort Procedures and Functions

(3) 14-48
```
function PutElement(var X) : Boolean;
  {-Put a sort element into the sort array}
```
(3) 14-47
```
function GetElement(var X) : Boolean;
  {-Get next element from the sorted array}
```
(3) 14-49
```
function Sort(Elements    : Word; ElementSize : Word;
              GetElements : GetPutProc; LessFunc : UserLessFunc;
              PutElements : GetPutProc) : SortStatus;
  {-Sort a group of elements}
```

Index

0

8087 reals 11-17
8x8 font 3-22, 35
<CtrlBreak> 4-152
{$N-} 2-9
{$V-} 6-112, 6-176

A

Action routines (for selectors) 6-85
Active coordinates (for windows) 4-70
Active window 4-56
Alternate draw 4-75
Array 8-154
Array, page structure 8-166
Array, priorities 8-157
Array, optimized 8-154
Array, performance 8-156
Array, range checking 8-158
ArrayField 6-29
ASCIIZ strings 9-35
Auto-advance 6-22
Auto-execution of commands 3-91

B

Background task hook 3-89
Backward links 6-74
BCD 14-2
BcdField 6-29
BEST 6-249
Binary Coded Decimal 14-2
Bit flags 14-28
Bit sets 8-50
Blinking attribute 3-6, 3-36
Block copy hook (for help) 4-411
Block function (for printer) 7-80
Block-oriented commands 5-1
BooleanField 6-30
Border 4-8, 4-21, 4-50
Box field 7-70
Boyer-Moore algorithm 9-4
BROWSE 5-80
BrowseStatusProc 5-63
Browser commands 5-57
Browsing 5-1
Burns, Philip R. 14-3
Buttons (see Mouse, buttons)
ByteField 6-30

C

C. Itoh 8510 printer 7-59
Calculated fields 6-124
Calculator 1-30, 14-3
Calendar 1-35
CALIBRAT.LZH 7-6
Calibration, printer 7-6
CGA 3-7
CharEditor 6-52, 6-119, 6-181
CharField 6-31
Character pitch 7-27
Character point 7-27
Characters per inch 7-27
CheckBreak 3-6
CheckEOF 3-6
CheckSnow 3-6
ChoiceField 6-31
Circular linked list 8-29
Cloning 14-16
Cluster size 4-292
Code numbers for objects 8-96
Color names 3-49
ColorMap 3-8
ColorSet 2-8, 3-45
ColorSet and streams 4-160
Commands, codes 3-83
Commands, general 4-140
Commands, keyboard handling 2-8
CompField 6-32
Compiling large programs 15-24
CompuServe 1-45, title-page
Conditional defines 2-9, 4-22, 4-57
Configuration data 2-8, 14-16
Constructors 2-5
Context sensitive help 2-8, 3-90, 4-138, 4-141, 4-312, 4-361, 4-399
Conventions (for this manual) 2-4
Conversion routine (for data entry) 6-15, 6-141
Conversion routine (for printer) 7-77
Copy file 10-6
Cross-reference (for help) 4-367
CRT 3-3
Current field (for data entry) 6-59
Current item procedure (for PickList) 4-312, 4-330
Custom string procedure (for PickList) 4-309
Customizing programs 14-16

D

Data entry 1-2, 6-1
Data entry, commands 6-136
Data entry, fields 6-15
Data entry, nested screens 6-127
Data entry, overview 6-140
Data entry, screens 6-110
Data objects 1-2, 8-1
DataProducts DP 7-59
Date variables 6-206
Date, valid ranges 6-207

DateField 6-32
DateStField 6-33
Decipoint 7-28
Derived types 2-4
DESKPOP 1-24
DesqView 3-1, 3-8
Destructors 2-5
Diablo daisywheel printer 7-27, 7-58
DirectVideo 3-6
Directory 10-18
Directory, changing 4-253
Directory, commands 4-243
Directory, entry (for libraries) 8-127
Directory, format 4-278
Directory, list 4-241
Directory, sorting 4-246
Directory, tree format 4-282, 4-293
Directory, window 4-281
Disk transfer address (DTA) 10-1
Disk class 10-13
Disk sectors 10-22, 10-31
Divider bars 4-39
DPINST 1-40
DOS critical error handler 11-16
DOS environment 10-1
DOS services 11-15
DOS services for printer output 7-4
DOS shell 10-27
DOS unit 10-1
DosBusyFlag 11-27
DosCriticalFlag 11-27
DoubleField 6-33
Doubly linked list 8-29
Draw, menu 4-306
Draw, window 4-72

E

EDITOR 5-79
Editor 1-33, 5-19
EGA 3-1, 3-7, 3-22, 3-35
EMS 8-154, 12-1, 12-3
Enhanced keyboards 3-5, 10-2, 10-42
ENTRY 6-246
Entry screens 6-1
Entry screens and streams 6-133
Entry screens, wrap mode 6-62, 6-80
EntryField 6-20
Epson LQ printers 7-58
Epson MX printers 7-58
Epson dot matrix printers 7-27, 7-58
Error categories 8-150
Error codes 4-141, 8-150, 15-10
Error handling 2-7, 4-140, 7-79, 8-150, 8-158
Error handling, default 4-156
Error message strings 8-150
Error types 15-10
Exchanging values 14-32
ExecDos 10-1
Exit command 6-141, 4-152, 4-157
Expanded memory (see EMS)
Exploding windows 4-77, 4-333
Extended memory 12-1
ExtendedField 6-33
Extension (file) 9-11, 9-13, 9-18
External Interfaces (for TSRs) 11-19, 11-41

F

Facit 4512 printer 7-59
Far calls 14-34
FastDispose 2-9
Field, adding 6-114
Field, attributes 6-58
Field, conversion routines 6-40, 7-76
Field, definitions 6-4
Field, delimiters 6-123
Field, derived types 6-29
Field, drawing routines 6-48
Field, editors 6-4, 6-15, 6-52, 6-141
Field, hidden 6-56, 6-58
Field, ID 6-55, 7-96
Field, invisible 6-57
Field, links 6-60
Field, lists 6-55
Field, movement commands 6-60, 142
Field, options (for data entry) 6-21, 6-122, 7-86
Field, protected 6-56, 6-58
Field, selecting 6-1, 6-3, 6-55, 6-58
Field, user-defined 6-130
Field, validation routines 6-43, 6-44
File attribute 4-242
File browser 5-1, 5-56
Filename 9-18
Filename filtering 4-244
Filename formatting 4-246
Flags 14-44
Float 9-2
Font 7-28, 7-32
Font, stack 7-32
Font, changing 7-27
Font8x8 3-6
Footers 7-78
Formatted strings 9-14, 9-21, 9-49
FORMTEST 7-113
FourByteDates 2-9
Frame 4-8, 4-20 (see Window)
Frame-relative writing 4-81, 4-114
Free pool, window numbers 4-129

G

Generating documentation 6-244

Generating menus 4-420
Generating source code 6-244
Global validation routine 6-126, 6-156
GRAPH3 3-3
GUIDE 1-17

H

Hashing 8-68
Header, more (for PickLists) 4-208, 4-384
Header, printer 7-78
Header, spanning 4-39
Header, window 4-21
Heap usage 2-7
Help 1-17, 4-361
Help, bound in 4-372, 4-394
Help, context-sensitive 2-8, 3-90, 4-138, 4-141, 4-312, 4-361, 4-399
Help, cut/paste 1-21, 4-411
Help, file 4-375
Help, file control characters 4-367
Help, file directives 4-364
Help, index 4-138, 4-365
Help, paged 4-406
Help, previous topic stack 4-377, 4-405
Help, scrolling text 4-409
Help, topics 4-298, 4-365
Hexadecimal string 9-16
HP DeskJet 7-58
HP LaserJet 7-27
Hot spot 4-25
Hot spot, for help 4-377
HotCode 4-91
Hotkeys 11-13
HugePtr 12-12

I

ICD Files 2-8
InitAllUnits 2-9
InitStatus 8-151
Inline macros 14-1, 28
Inline procedures 13-1
INSTALL.EXE 1-10
Installation 1-7, 1-10
IntegerField 6-34
International currencies/conventions 6-11
Interrupt service routines 11-1
Interrupt vector 11-1, 11-4
Interrupts and TSRs 1-3
Item key (for menus) 4-298

J

Julian date 6-208
Jump marker 14-28

K

Key assignment 3-87, 3-112
Key codes 10-42
Keyboard macros 1-3, 13-1
Keyboard status flags 3-27
Keystrokes, commands 3-82

L

LHARC 1-5, 14-28
Landscape mode 7-28
Large array (see Array)
LaserJet printers 7-33, 7-58
LastMode 3-6
Leading 7-28
Letter sets 9-8
Library 8-127
Library directory 8-127
Line editor 6-1, 6-176
Linked list 8-29
Literals (see Picture masks)
LoadableColorSet 4-2
LongIntField 6-34
Lotus style menus 4-297

M

Macro, chaining 13-12, 13-21
Macro, editor 13-15
Macro, engine 13-1, 13-8
Macro, file 13-11, 23
Macro, nested 13-1
Macro, playback 13-18
Macro, recording 13-10
Macro, scrap 13-18
Make file (installation) 1-7
MAKEHELP 4-369
MAKEMENU 4-420
MAKESCRN 6-228
Mapping, video attributes 3-28
Mask (see Picture mask)
Mask, characters 6-5
Mask, date 6-6
Mask, directory header 4-251
MCGA 3-7
Memo 5-1, 5-79
Memo field 5-2, 5-6
Memo window 5-22
Memory allocation 1-3, 12-1
Memory resident programs (see TSR)
Menu, building 4-298
Menu, commands 4-304
Menu, customization 4-301
Menu, Lotus style 4-297
Menu, protected items 4-203, 4-308
Menu, systems 4-297
MergePicture 6-14

Methods, static 2-4
Methods, virtual 2-4
Mickeys 3-60
Micronics printer 7-58
MODE 7-5
Modules (installable) 11-19
Mono 3-6
MonoHerc 3-7
Mouse, buttons 3-54, 3-68
Mouse, coordinates 3-55
Mouse, cursor 3-53
Mouse, drivers 3-59
Mouse, droppings 3-56
Mouse, event Handlers 3-57
Mouse, support 2-8, 3-1, 3-88
Mouse, windows 3-55
MultiLineField 6-35, 6-125, 6-128
Multiple choice lists 4-196
Multitasking system 3-8

N

Naming conventions (for this manual) 2-6
NEC 3510/3530 printers 7-58
NEC 7710/7730/8810/8830 printers 7-59
NestedField 6-35
NullValidation 6-44
NumberEditor 6-52, 120, 181
Numeric coprocessor 11-3
Numeric field 6-21

O

Object Professional Library 8-129
Object, code numbers 8-96
Object, hierarchy 15-7
Object, version number 8-96
Object-oriented I/O 8-90
Object-oriented data structures 8-2
Octal string 9-23
OmniView 3-1, 3-8
OPL 8-129
OS 14-28
Overlays 11-17, 15-14

P

Packed image 4-165
Packed window 4-436
Packed window, bound in 4-168
PackedWindow 4-164
Page frame 12-7 (see EMS)
Panasonic KX-P1093 printer 7-58
Pasting help text 1-21
Path list 4-241
Pathname 9-7, 9-16, 9-19
Periscope 11-16
PGC 3-7
PickHorizontal 4-194
PickList 4-186
PickList field 6-128
PickList, orientations 4-194
PickList, user move hook 4-203
PickListFields 2-10
PickSnaking 4-194
PickVertical 4-194
Picture mask (for printing) 7-2
Picture mask 6-4, 141, 241
Picture mask, semi-literal characters 6-5
Picture mask, user defined characters 6-6
Pitch, character 7-27
PlaybackDelay 13-3
Point 7-28
Point, character 7-27
Pointer arithmetic 14-1
PointerStack 8-6
POPHELP 1-17, 11-35, 13-23
Popup, dispatcher 11-13
Popup, handles 11-14
Popup, routines 11-13, 14
Portrait mode 7-28
Post-edit routines 6-41, 6-124, 6-141, 6-150
Post-select routines 6-86
Pre-edit routine 6-113, 6-123, 141
Pre-select Routines 6-86
PRINT.COM 10-1
Printing 1-2, 7-1
Printing, calibration 7-6
Printing, custom hook
Printing, error handling 7-1, 7-34
Printing, fields 7-2
Printing, forms 7-66
Printing, hardware registration 7-57
Printing, lines and boxes 7-27
Printing, modes 7-2
Printing, overview 7-84
Printing, page function 7-80
Printing, printer libraries 7-59
Printing, streams 7-35, 7-81
Printing, user data field 7-98
Printing, user hooks 7-79
Procedure pointers 2-4
Program customization 14-16
Proprinter II printer 7-58
Protected field 6-56
PSCREEN 4-436
Pseudo-scan code 3-71
PTIME 11-64
Pulldown menus 4-297
Purchase Agreement 1-45

Q

Queues 8-19
Quick Reference Chart 1-38

Quicksort 14-1

R

Range checking (large arrays) 8-158
READ.ME 1-5
Reading/writing text files 5-19
RealField 6-36
Regular field 6-21
Reject functions (for DirList) 4-245
Replay 13-2
Reports 7-68
Required fields 6-126
Run-length compression 4-165
Runtime error recovery 14-25
Runtime errors 11-16

S

SaveInt1B 3-6
Saving, windows 4-112
Scan code 3-31
Scrap macro 13-18
Screen rectangles 4-9
Screen update routines 6-61
Screen/keyboard Handling 1-1
Scroll bars 4-21, 4-25, 4-113
Scrolling, windows 4-112
Search attribute (for files) 4-274
Search mask (for files) 4-242, 4-274
Search string header (for files) 4-209
Search-and-replace 5-1
Secondary command tables 3-90
Secondary field options 6-24
SelectField 6-17
Selecting fields 6-3, 6-58
Selecting fields, Action Routines 6-85
Semi-literal characters 6-5
Separator bar 4-328
Shadows 4-21, 24
ShortIntField 6-36
SideKick 11-21
Simple field 6-21
Simple line editor 6-197
SimpleCharEditor 6-52, 6-120, 6-179
SimpleStringEditor 6-52, 6-118, 6-177
SingleField 6-37
Singly linked list 8-29
Site licensing 1-45
Slider 4-32
SMACS 13-23
Sorting 14-45
Soundex 9-26
StackWindow 4-5
Stacks 8-6, 11-1, 11-15
Standard TSR interface 11-19
Static methods 2-4
StaticQueue 8-19
Status byte (for printing) 7-5
Stream 2-7, 8-90
Stream manager 8-94
Stream, memory 8-94
Stream, registration routines 6-50, 8-94
Strenczewilk, Don 9-5
String dictionary 8-67
String handling 1-2, 9-1
String management 8-67
String searching 4-198
StringArray 8-67
StringEditor 6-52, 6-116, 6-179
StringField 6-37
StringSet 8-67
StripPicture 6-14
Subfields 6-5, 8
Swap region 11-2
Swapping, TSR 11-2, 11-34
Swapping, disk/EMS 11-40
Swapping, exec manager 10-32
Swapping, messages 11-40
System requirements 1-4

T

Tab characters 5-20, 9-11
TaskView (see OmniView)
Technical support line 1-46
Terminate and Stay Resident 11-1
Text editor 1-2, 5-1, 5-3
Text editor, memo fields 5-37
Text file device drivers 7-8
TextAttr 3-6
TextField 6-15
ThwartSideKick 2-10
Time variables 6-206
TimeField 6-37
Toshiba P321SL or P341SL printers 7-58
TSR 11-1
TSR, unloading 11-20
Turbo Professional, converting from 15-16
TURBO3 3-3
TURBO5.MAC 13-23
Typecasting 8-31, 14-28

U

Unit dependencies 15-4
Unloading a TSR 11-20
UseAdjustableWindows 2-10
UseBCD 2-10
UseCrt 2-10
UseDates 2-10
UseHotSpots 2-10
UseMouse 2-10
UseScrollBars 2-10

UseShadows 2-10
User exit commands 3-95

V
Validation (see Fields)
VDISK.SYS 12-1
VGA 3-1, 3-7, 3-22, 3-35
Video attribute 3-1, 3-4, 3-48
Video memory 3-29
Video page 3-1, 3-24, 3-36, 3-39, 3-66, 3-79
Video segment 3-24, 4-173
Virtual methods 2-4
Virtual screen 4-173, 6-101

W
Warranty period 1-45
WFrame 4-21
White space 9-31
WindMax 3-6
WindMin 3-6
Window 4-8
Window, coordinates 3-33, 4-55
Window, headers 4-23
Window, hierarchy 4-1, 4-55, 4-138
Window, manager 4-56, 4-112
Window, mixing RawWindow and
StackWindow 4-116
Window, moving and resizing 4-113
Window, objects 1-1, 4-1, 4-8
Window, saving 4-112
Window, scrolling 4-112
Window, separator bar 4-328
Window, shadows 4-21, 24
Window, sizing for PickLists 4-195
Window, stack 4-112, 15-15
Window, streams 4-57
Window-relative writing 4-103, 4-113
WINWOW 4-438
Wrap mode 6-62, 6-80
Word wrap 9-34
WordField 6-38

Y
YesNoField 6-38

Z
Zero-based arrays 8-158

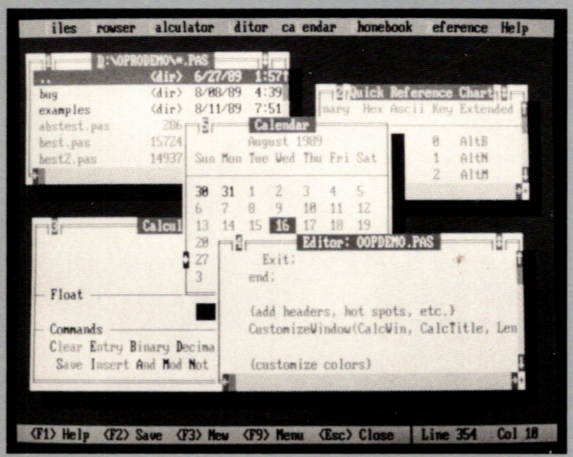
Object Professional™

A Powerful Library for Object-Oriented Programming in Pascal

Object-oriented programming multiplies your productivity by promoting reuse of proven software modules. Don't start from scratch with your objects! Use Object Professional 1.0, a huge library of over 100 powerful object types containing over 1000 methods.

Powerful User Interface Objects

The window objects let you use multiple overlapping and resizeable windows. Your windows will provide capabilities like:



- menus
- pick lists
- file browsing
- scrolling data entry screens with:
  - help capability
  - line editors
  - data validation and formatting
  - hidden and protected fields
  - matching capability for printed forms
- mouse support and scroll bars
- text editing

The window objects are completely flexible—you can create text-mode PM look-alikes or your own unique look and feel. Includes source generator for menus and entry screens.

Data Object Types . . . and System-Oriented Routines Too !

Build your programs from proven and documented object types like stacks, linked lists, virtual arrays, and more. System-oriented routines provide: ■ Swappable TSRs in only 6K of RAM ■ Interrupt management ■ EMS management ■ DOS and BIOS capabilities, and more.

Documentation: Clear and Complete

Object Professional includes clear, comprehensive documentation, on-line help, full source code, technical support, and hot demo programs. You pay no royalties when you use these routines.

Object Professional requires Turbo Pascal 6.0 or 5.5, PC-DOS or MS-DOS 2.0 or later, and an IBM PC, XT, AT, PS/2 or compatible.

TURBO Power

TurboPower Software
P.O. Box 49009
Colorado Springs, CO 80949-9009
CompuServe I.D. 76004, 2611

©TurboPower Software 1991